공학박사학위논문

실행시간 프리페칭 기법을 이용한 응용프로그램의 기동시간 단축

Reducing Application Launch Time by Using Execution-time Prefetching Techniques

2013년 2월

서울대학교 대학원

전기·컴퓨터공학부

유 준 희

실행시간 프리페칭 기법을 이용한 응용프로그램의 기동시간 단축

Reducing Application Launch Time by Using Execution-time Prefetching Techniques

지도교수 신 현 식

이 논문을 공학박사 학위논문으로 제출함

2012년 12월

서울대학교 대학원

전기·컴퓨터공학부

유 준 희

유준희의 공학박사 학위논문을 인준함

2013년 2월

위 원 장 _____ (인)

부 위 원 장 _____ (인)

위 원 _____ (인)

위 원 _____ (인)

위 원 _____ (인)

학위논문 원문제공 서비스에 대한 동의서

본인의 학위논문에 대하여 서울대학교가 아래와 같이 학위논문 저작물을 제공하는 것에 동의합니다.

1. 동의사항

① 본인의 논문을 보존이나 인터넷 등을 통한 온라인 서비스 목적으로 복제할 경우 저작물의 내용을 변경하지 않는 범위 내에서의 복제를 허용합니다.

② 본인의 논문을 디지털화하여 인터넷 등 정보통신망을 통한 논문의 일부 또는 전부의 복제·배포 및 전송 시 무료로 제공하는 것에 동의합니다.

2. 개인(저작자)의 의무

본 논문의 저작권을 타인에게 양도하거나 또는 출판을 허락하는 등 동의 내용을 변경하고자 할 때는 소속대학(원)에 공개의 유보 또는 해지를 즉시 통보하겠습니다.

3. 서울대학교의 의무

① 서울대학교는 본 논문을 외부에 제공할 경우 저작권 보호장치(DRM)를 사용하여야 합니다.

② 서울대학교는 본 논문에 대한 공개의 유보나 해지 신청 시 즉시 처리해야 합니다.

논문제목 : 실행시간 프리페칭 기법을 이용한 응용프로그램의 기동시간 단축

학위구분 : **석사 □·박사 ☑**

학 과(부) : 전기.컴퓨터공학부

학 번 : 2005-30312

연 락 처 : 02-880-7298

저 작 자 : 유 준 희 (인)

제 출 일 : 2013 년 2 월 1 일

서울대학교총장 귀하

초 록

최근 모바일 기기의 사용이 보편화되면서 프로그램 실행의 응답성은 사용자의 체험에 큰 영향을 주는 요소가 되었다. 특히, 응용프로그램의 기동시간은 기기에 대한 사용자의 체감성능을 평가하는 중요한 지표로 사용된다. 하지만 플래시 기반의 디스크가 시스템 디스크로 사용되는 경우에도 사용자들은 긴 응용프로그램 기동시간을 자주 경험한다. 프로세서나 디스크 장치는 병렬성을 이용하여 성능을 개선하는 반면, 응용프로그램 기동 시에는 자원들의 사용이 직렬화되기 때문이다.

응용프로그램의 기동시간 단축을 위하여 본 논문에서는 새로운 실행시간 프리페칭 기법을 제안한다. 응용프로그램의 최초 기동 시 접근되는 블록을 정확히 파악하고, 이후 기동 시 이 블록들을 효율적인 방법으로 디스크캐시에 적재함으로써 기동시간을 단축시킨다. 핵심 전략은 프로세서와 디스크의 사용을 병렬화하고 디스크의 내부병렬성과 멀티코어의 활용을 유도하였다. 또, 프리페칭 시간을 단축하기 위하여 디스크의 특성에 따라 다양한 병합, 논리블록번호 정렬, 프리페치 수준의 의존성 해결 기법을 사용하였다.

제안한 프리페칭 기법을 리눅스 커널 3.5.0에 구현하였고 많이 사용되는 응용프로그램을 이용하여 성능을 평가하였다. 하드디스크 기반의 데스크탑 워크로드에서 콜드스타트 시간 대비 평균 52%의 기동시간을 단축하였고 SSD를 사용한 경우 34.1%가 단축하였다. 또, SSD를 사용하는 모바일용 Meego 플랫폼에서 평균 28.1 ~ 34.1%의 기동시간을 단축하였고, 안드로이드 플랫폼이 탑재된 갤럭시 넥서스 폰에서 평균 12.8%의 기동시간을 단축하였다. 마지막으로, 유저 수준에서 구현한 프리페쳐를 사용한 경우, SSD를 사용하는 환경에서 평균 21.7 ~ 28.5%의 기동시간을 단축하였다.

제안한 기법을 기존의 환경에 구현하여 운용하는데 경미한 오버헤드를 유발하는 한편, 시스템의 응답속도를 개선하고 사용자의 체감 속도를 향상시킴으로서 데스크탑 PC와 스마트폰의 성능향상에 유의미한 기여를 할 것이다.

주요어: 응용프로그램 기동, 기동시간 단축, 디스크 입출력 최적화, 실행시간 프리페쳐, 응용프로그램 프리페쳐

학 번: 2005-30312

목 차

제 1 장 서 론 ····· 1

1.1 연구 동기 ····· 1
1.2 연구 내용 및 의의 ····· 3
1.3 논문의 구성 ····· 8

제 2 장 연구 배경 ····· 9
2.1 범용 디스크 드라이브 ····· 9
2.1.1 하드디스크 드라이브 ····· 9
2.1.2 NAND 플래시 기반 Solid-State Drive (SSD) ····· 10
2.1.3 하이브리드 하드디스크 ····· 12
2.2 리눅스의 디스크 입출력 부 시스템 ····· 14
2.2.1 리눅스의 디스크 입출력 스택 ····· 14
2.2.2 리눅스의 디스크캐시 ····· 16
2.2.3 입출력 스케쥴러의 종류 및 특징 ····· 18
2.2.4 입출력 플러그/언플러그 ····· 20
2.2.5 프리페치 성공 시 절약되는 프로세서 시간 분석 ····· 21
2.3 응용프로그램의 빠른 기동을 위한 기존 연구 ····· 23
2.3.1 응용프로그램의 빠른 기동을 위한 디스크캐싱 기법 ····· 23
2.3.2 범용 워크로드의 빠른 응답을 위한 디스크캐싱 기법 ····· 26
2.3.3 그 외의 기법들 ····· 29

제 3 장 응용프로그램 기동 시의 동작 특성 분석 ····· 31
3.1 기동 시나리오 ····· 31

3.2 기동 시 발생되는 디스크 입출력 분석 ····· 32
3.3 프로세서와 디스크의 활성화 패턴 분석 ····· 34

제 4 장 커널 수준 실행시간 프리페쳐의 설계, 구현 및 평가 ····· 37
4.1 실행시간 프리페쳐의 소개 및 목표 ····· 37
4.2 기동시퀀스 수집 ····· 41
4.3 프리페치시퀀스 스케쥴러 ····· 44
4.3.1 익스텐트-의존성 (Extent-Dependency) 분석 ····· 44
4.3.2 블록 간 의존성 해결을 위한 메타데이터 쉬프트 ····· 46
4.3.3 거리기반 병합 ····· 50
4.3.4 거리기반 빈공간채움 병합 ····· 51
4.3.5 논리블록번호 정렬 ····· 52
4.3.6 플러그/언플러그 ····· 52
4.4 응용프로그램과 프리페쳐 동작의 병렬화 ····· 54
4.4.1 하드디스크를 사용하는 시스템 ····· 54
4.4.2 SSD를 사용하는 시스템 ····· 56
4.4.3 다중 디스크를 사용하는 시스템 ····· 57
4.5 기동시퀀스의 유효성 관리 ····· 57
4.6 운영체제 부트 프리페쳐 ····· 58
4.7 유휴시간 프리페쳐 인터페이스 ····· 59
4.8 실험 환경 ····· 60
4.9 응용프로그램 기동시간 ····· 64
4.10 운영 및 저장 공간 오버헤드 ····· 78
4.11 커널 수준 프리페쳐의 안전성 ····· 80

제 5 장 유저 수준 실행시간 프리페쳐의 설계, 구현 및 평가 ····· 83
5.1 유저 수준 프리페쳐의 소개 및 구조 ····· 83
5.2 응용프로그램의 프리페치시퀀스 생성 ····· 85
5.2.1 디스크 입출력 정보 수집 ····· 85
5.2.2 기동시퀀스 추출 ····· 85
5.2.3 프리페치시퀀스 스케쥴 ····· 86
5.3 블록-파일 사상 (Map) ····· 86
5.3.1 블록-파일 사상의 소개 ····· 86
5.3.2 기동시퀀스 관련 파일 목록의 수집 ····· 88
5.4 유저 수준의 프리페쳐 프로그램 생성 ····· 89
5.5 응용프로그램 기동 관리자 ····· 90
5.6 유저 수준의 프리페쳐의 장점 및 단점 ····· 93
5.7 실험 환경 ····· 93
5.8 응용프로그램 기동시간 ····· 94
5.9 운영 및 저장 공간 오버헤드 ····· 95

제 6 장 결론 및 향후 연구 방향 ····· 96
6.1 결론 ····· 96
6.2 향후 연구 방향 ····· 99

참고문헌 ····· 102

Abstract ····· 113

표 목차

표 1 응용프로그램 기동 시 읽기 및 쓰기 I/O 요청의 개수와 총 크기 33
표 2 프리페치 단계 ····· 40
표 3 기동시퀀스 로그 정보 ····· 42
표 4 기동시퀀스 유효성 관리를 위한 파라미터 ····· 58
표 5 수집된 기동시퀀스의 입출력 정보 (SSD) ····· 64
표 6 수집된 기동시퀀스의 입출력 정보 (HDD) ····· 65
표 7 빈공간 병합에서 빈 공간 허용치 증가에 따른 입출력 수 ····· 69
표 8 빈공간 병합에서 빈 공간 허용치 증가에 따른 입출력 총 크기 ·· 70
표 9 거리와 빈 공간 허용치에 따른 프리페치 요청의 크기 ····· 75
표 10 거리와 빈 공간 허용치에 따른 프리페치 요청의 수 ····· 75
표 11 64비트 데스크탑 워크로드의 프리페치 정보 파일의 크기 ····· 79
표 12 블록 종류에 따른 블록 캐싱 시스템 콜 ····· 89
표 13 응용프로그램 기동 관리자가 사용하는 설정 값 및 변수 ····· 92

그림 목차

그림 1 SSD의 내부구조 ····· 11
그림 2 리눅스의 디스크 입출력 처리 흐름 ····· 14
그림 3 디스크 요청에 대한 처리 시간 요소 ····· 21
그림 4 Eclipse 기동 시 프로세서와 SSD의 사용 패턴 ····· 34
그림 5 Eclipse 기동 시 SSD에서 동시에 처리 중인 요청의 개수 ····· 34
그림 6 응용프로그램의 기동 시 발생하는 블록 요청의 크기별 비율 ·· 35
그림 7 제안한 실행시간 프리페쳐의 프레임워크 ····· 37
그림 8 프리페치 단계 전이도 ····· 38
그림 9 기동시퀀스 로깅이 수행되는 함수와 디스크 입출력 계층 ····· 42
그림 10 익스텐트-의존성 분석 ····· 44
그림 11 파일시스템 수준의 의존성 ····· 47
그림 12 프리페치 수준의 의존성 ····· 48
그림 13 메타데이터 쉬프트를 이용한 프리페치 의존성 해결 ····· 49
그림 14 거리기반 병합 ····· 50
그림 15 거리기반 빈공간 채움 병합 ····· 51
그림 16 하드디스크를 위한 동기 프리페칭 ····· 54
그림 17 하드디스크를 위한 동기, 비동기 하이브리드 프리페칭 ····· 55
그림 18 SSD를 위한 비동기 프리페칭 ····· 56
그림 19 응용프로그램의 기동시간 (HDD) ····· 66
그림 20 응용프로그램 기동에 소비되는 에너지 분석 (HDD) ····· 67
그림 21 빈공간 병합을 통한 기동시간 최적화 (HDD) ····· 68
그림 22 동기, 비동기 하이브리드 프리페칭을 이용한 기동시간 단축 · 71

그림 23 응용프로그램의 기동시간 (SSD) ····· 72
그림 24 Meego 용 응용프로그램의 기동시간 ····· 73
그림 25 Meego 용 응용프로그램의 기동시간 (화면 전환 적용) ····· 74
그림 26 안드로이드 용 응용프로그램의 기동시간 ····· 77
그림 27 FAST의 프레임워크 ····· 83
그림 28 응용프로그램의 기동시간 (삼성 SLC SSD) ····· 94
그림 29 응용프로그램의 기동시간 (OCZ Nocti SSD) ····· 94

제 1 장 서 론

1.1 연구 동기

응용프로그램의 기동시간 (launch time)은 데스크탑 PC나 모바일 장치의 사용자 체감성능을 평가하는 중요한 지표로 사용되어 왔다. 특히, 스마트폰과 같이 항상 휴대하면서 자주 이용하고 짧게 사용하는 장치에서는 응용프로그램의 기동시간 단축이 매우 중요하다 [26][33][34]. 하드웨어의 성능이 끊임없이 향상되고 있음에도 불구하고 기동시간은 하드웨어의 향상된 성능만큼 단축되지 않는다. 왜냐하면 응용프로그램 역시 새로운 버전이 출시될 때 마다 확장된 기능을 준비하기 위하여 더 많은 연산과 디스크 처리를 요구하기 때문이다 [25]. 이 때문에 기동시간을 단축시키기 위하여 소프트웨어 및 하드웨어 수준에서 많은 기법들이 연구되었다. 예를 들면 기동에 불필요한 처리를 기동 이후로 연기하거나 플래시 저장 장치를 하드디스크의 캐시로 이용하여 느린 하드디스크의 성능을 보완하는 방법 등이 있다 [10][11][15][17][20].

소프트웨어 수준의 기동시간 단축 기법이 중요한 이유는 응용프로그램 기동에 가장 큰 영향을 주는 프로세서와 디스크 장치의 성능 개선 기법이 기동속도 향상에 크게 기여하지 못하는데 있다. 예를 들면, 응용프로그램이 기동될 때 프로세서가 활성화되는 시간의 대부분에서 하나의 코어만이 사용된다. 하지만 최근 프로세서의 성능을 향상시키는 주된 기술은 멀티코어 기법이며 단일 코어의 성능은 느리게 향상되고 있다. 그리고 응용프로그램 기동 시에 발생하는 디스크 요청들은 대부분 작은 크기의 임의 접근 패턴을 보인다. 하드디스크의 경우에는 집적도 향상으로 순차 처리 성능은 향상되고 있지만 임의 접근 패턴의 처리 성능에 영향

을 많이 주는 탐색 시간 (seek time) 및 회전 지연 (rotational latency)은 거의 향상되지 못하고 있다. 2000년대 후반부터 많이 사용되기 시작한 NAND 플래시 기반 solid-state drive (SSD)는 기계적인 움직임이 없어 매우 빠른 접근 속도를 보이며 여러 개의 NAND 칩을 병렬적으로 작동시켜 단일 칩의 처리율 한계를 극복해 왔다 [3][29][53]. 하지만 응용프로그램 기동 중에 발생하는 디스크 요청의 대부분은 작은 크기의 요청이고, 이는 SSD의 칩 수준의 병렬성을 심각하게 제약한다. 단일 칩의 성능은 느리게 개선되며 이론적으로 40MB/s 까지 가능하나 실제 제품에서는 이에 훨씬 못 미치는 성능을 보여준다. 따라서 SSD를 사용하는 시스템에서도 디스크 시간의 비중이 작지 않다 [25][36][37].

마이크로소프트의 윈도우즈와 애플의 Mac OS 운영체제에서는 시스템 부팅이나 응용프로그램의 기동시간을 단축시키기 위하여 다양한 프리페칭 기법들을 커널 (kernel)에 탑재하였다 [43][55][58]. 리눅스의 경우에는 커널에 공식적으로 탑재된 것은 없지만 유저영역에서 동작하는 유휴시간 프리페치 프로그램이 있다 [6]. 실행시간 프리페쳐는 운영체제나 응용프로그램이 처음 실행될 때 접근되는 블록들을 기록하였다가 이후에 실행될 때 수집된 정보를 통하여 블록들을 효율적인 순서로 디스크 캐시에 미리 읽음으로써 응용프로그램의 기동시간을 단축시키는 소프트웨어 기법이다. 이 기법은 추가적인 하드웨어 장착을 필요로 하지 않고 프로그램의 소스코드나 바이너리의 수정 등을 필요로 하지 않기 때문에 기동시간 단축을 위한 가장 현실적인 해결 방법 중 하나로 여겨진다.

기존의 실행시간 프리페칭 기법은 하드디스크를 사용하는 환경에서 디스크 헤드의 움직임을 줄임으로써 기동시간을 단축시키는 방식을 사용해 왔다. 하지만 SSD와 같은 플래시 기반의 디스크는 기계적인 움직임

을 필요로 하지 않기 때문에 기존의 프리페칭 기법은 효과가 거의 없다. 하드디스크를 SSD로 교체하면 기동시간은 크게 단축되지만 여전히 디스크 시간은 기동의 큰 부분을 차지한다. 실제로 사용되고 있는 기존의 프리페쳐들은 기동시간 단축의 정도가 낮고 최근 많이 사용되고 있는 플래시 기반의 장치에서는 효과가 거의 없다. 따라서 데스크탑 PC나 스마트 기기에서 많이 사용되는 저장 장치인 하드디스크, SSD, 하이브리드 하드디스크, 그리고 이 장치들의 조합으로 구성된 디스크 상에서 효율적으로 동작하는 실행시간 프리페칭 기법에 대한 연구가 필요하다. 이를 위해 기존의 하드디스크 기반의 시스템에서 단순히 파일 수준의 정렬을 이용한 프리페칭 기법이 아니라 다양한 범용 디스크의 특성과 응용프로그램의 기동 특성을 면밀히 분석하고 이를 바탕으로 최소한의 운영 오버헤드로 기동시간을 최소화하는 기법의 연구 및 개발이 중요하다.

1.2 연구 내용 및 의의

본 연구는 데스크탑 PC부터 스마트폰에 이르는 다양한 개인용 기기에서 응용프로그램의 기동시간을 단축시키기 위한 실행시간 프리페칭 기법에 대한 내용을 다루며, 목표는 운영 오버헤드를 최소화하고 기동시간을 최대한 단축하는데 있다. 목표를 실현하기 위해서는 응용프로그램 기동 때 읽히는 블록들 (기동시퀀스)을 빠짐없이 파악하고 이후에 같은 프로그램이 다시 기동될 때, 수집된 기동시퀀스를 디스크캐시에 적재할 수 있어야 한다. 다시 말해, 수집된 기동시퀀스 정보는 디스크캐싱이 가능한 형태의 정보로 변경되어야 한다. 저 수준의 블록 접근을 관찰하여 기동시퀀스를 정확히 파악할 수 있지만 페이지캐시에 적재하기 위해서는 파일의 아이노드 번호와 파일 내 블록 번호가 필요하다. 따라서 디스크

캐싱에 이용할 수 있는 정보를 수집하여야 한다. 또, 프리페칭 시에는 대상 프로그램의 기동시퀀스를 최대한 빠르게 디스크캐시에 적재하는 동시에 프리페칭 작업과 대상 프로그램이 병렬적으로 수행되는 부분을 최대화하여야 한다. 마지막으로, 프리페칭 플랫폼의 운영에 필요한 계산 및 저장공간 오버헤드를 최소화하는 것이 중요하다. 연구 내용을 요약하면 다음과 같다.

응용프로그램의 기동시퀀스를 얼마나 정확히 수집해 내느냐는 프리페칭의 성능을 결정짓는 중요한 사항이다. 기동시퀀스의 감시는 다양한 수준에서 수행이 가능하다. 예를 들어, blktrace와 같은 도구를 이용하여 블록 수준의 정보를 수집하게 되면 수집되는 로그의 크기는 크지 않지만 디스크 캐싱을 위하여 파일 수준의 정보로 역변환하는 복잡한 과정이 필요하다 [25][37][38]. 반면에 디스크캐시에 내용이 있는지 검사하는 부분에서 기동시퀀스 정보를 생성하면 디스크캐싱을 하기 위한 정보로 변환이 쉽게 되지만 로그의 크기가 엄청나게 커진다. 이 외에도 디스크 입출력의 여러 계층에서 기동시퀀스의 수집이 가능하지만 본 논문에서 제안한 방법은 디스크캐시에서 검색 실패로 인해 실제로 하위 계층으로 디스크 입출력 요청을 내려 보낼 때 요청 관련 정보를 수집하여 기동시퀀스 정보를 생성한다. 이 방식은 로그의 크기를 작게 유지하면서 디스크캐싱을 위한 정보로 쉽게 변환이 가능하다.

기동시퀀스가 수집된 이후 해당 응용프로그램이 다시 수행되면 프리페칭을 통하여 기동속도를 빠르게 할 수 있다. 이 때, 디스크의 종류와 특성에 따라 기동시퀀스 수집 이후에 요청 순서를 최적화해 두어야 프리페칭 속도가 빨라져 실행중인 프로세스가 블록을 요청하였을 때 디스크캐시에서 읽어갈 확률이 높아진다. 이를 위해서 본 논문에서는 하드디스

크, SSD 등의 저수준 입출력 성능 분석과 파일시스템의 디스크 사용 특성 분석하여 프리페칭 성능을 최적화에 반영하였다. 하드디스크 장치에서는 헤드의 움직임을 최적화하고 플래시 기반의 SSD에서는 내부병렬화가 극대화되도록 다양한 병합 및 순서 변경 기법을 설계하고 구현하였다. 이러한 기동시퀀스의 스케쥴링은 기동시퀀스가 수집된 직후 수행되어 프리페칭 시에 최적화된 순서로 블록들이 프리페칭된다.

응용프로그램 기동 시에 프리페칭을 수행하는 방식은 프리페칭이 끝난 후 응용프로그램을 수행하는 동기 (synchronous) 프리페칭과 프리페칭을 위한 별도의 쓰레드를 생성하여 응용프로그램과 동시에 수행시키는 비동기 (asynchronous) 프리페칭이 있다. 하드디스크를 사용하는 시스템에서는 디스크 헤드의 움직임을 줄이기 위하여 논리블록번호로 정렬된 순서로 입출력을 요청하는데, 프리페칭 작업과 응용프로그램의 입출력 요청 간의 경쟁으로 인해서 발생될 수 있는 큰 디스크 헤드 움직임을 방지하기 위해서 동기프리페칭 기법을 사용한다. 하지만 SSD와 같이 기계적인 움직임이 없는 장치에서는 수집된 기동시퀀스의 순서대로 비동기 프리페칭을 수행하는 것이 좋다. 하드디스크 기반의 장치에서도 일부 블록은 비동기로 프리페칭하여 기동시간을 추가적으로 단축시킬 수 있다. 기동시간의 단축은 기동시퀀스 프리페칭의 성능 뿐 아니라 대상 프로세스와 프리페쳐의 병렬성도 중요하다. 본 논문에서는 내부병렬성 이용을 통하여 기동시퀀스를 빠르게 프리페칭하는 기법, 응용프로그램의 수행과 디스크 프리페칭 작업의 병렬성을 높이는 기법을 함께 고려하여 응용프로그램의 기동시간을 단축시켰다.

응용프로그램의 기동시퀀스는 시간에 따라 변할 수 있다. 예를 들면, 라이브러리나 프로그램을 업데이트 하는 경우 또는 사용자 데이터의 변

경 등은 동적으로 데이터가 변경되는 경우를 들 수 있다. 또, 기동시퀀스 수집 중 관련 없는 프로세스로부터 발생한 디스크 요청 정보가 기동시퀀스에 포함되어 있을 수 있다. 기동과 상관없는 블록들을 기동시퀀스에서 제거하고 필요에 따라 기동시퀀스를 재생성하기 위한 과정들이 프리페쳐 프레임워크 안에 포함되어야만 기동시퀀스의 높은 정확도를 유지할 수 있다. 따라서 이러한 작업을 위한 효과적이고 오버헤드가 적은 기법을 설계하고 구현한다.

일반적으로 프리페쳐의 성능을 결정짓는 외부 요소는 프로세서와 디스크의 성능이다. 본 논문에서는 제시한 프리페쳐의 성능을 데스크탑 PC 환경에서 평가하기 위하여 인텔 I7-2620m 프로세서, 7200RPM의 하드디스크 또는 SSD가 장착된 노트북에서 실험을 수행하였다. 사용된 벤치마크 용 프로그램은 Fedora 16 64비트 배포판에 포함되어 있는 응용프로그램 9개와 많이 사용되는 외부 응용프로그램 5개를 이용하였다. 실험 결과, 하드디스크를 이용한 경우 콜드스타트 대비 52%, SSD를 사용한 경우 34.1%. 하드디스크를 이용한 실험 결과는 기존의 구글-프리페치 기법보다 콜드스타트 대비 22% 단축된 수치이다.

그 외에도 모바일 플랫폼에서 프리페쳐의 성능을 분석하기 위하여 인텔 듀얼코어 Atom 1.83GHz 프로세서와 느린 SSD가 장착된 태블릿 PC에 Meego 1.2 태블릿 버전을 설치하고 Meego 용 응용프로그램 6개를 이용하여 모바일 플랫폼에서의 프리페쳐 성능을 측정하였다 [59]. 실험 결과 화면 전환 여부에 따라 평균 28.1 ~ 31.4%의 기동시간이 단축되었다. 또, 최근 모바일폰에서 가장 많이 사용되는 안드로이드 플랫폼에서 제안한 프리페쳐의 성능을 평가하기 위하여 삼성 갤럭시 넥서스폰에서 안드로이드 용 응용프로그램 9개를 이용하여 기동시간을 측정하였

다. 제안한 프리페쳐를 이용한 경우 12.8%의 기동시간이 단축되었다.

본 논문의 의의는 다음과 같다.

1. 응용프로그램 기동 시 시스템 자원의 사용 패턴과 동작 특성을 분석하였다. 기동에 가장 큰 영향을 주는 시스템 자원인 프로세서와 디스크의 활성화 패턴을 분석하고, 이를 통하여 기동시간 단축을 위한 접근 방법을 설명하였다. 이는 추후 기동시간 단축 또는 프리페칭 기법 연구에 큰 도움이 된다.

2. 응용프로그램의 기동시퀀스를 재현가능한 수준의 정보로 빠짐없이 수집하는 방법을 제안하였다. 제안한 기법은 디스크캐시 무효화 후, 버퍼캐시 및 페이지캐시 미스로 인해 발생되는 입출력 정보를 수집하였고 파일시스템 수준의 의존성 분석을 통하여 무효화에 실패한 기동관련 블록을 파악하여 기동시퀀스 수집 정확도를 높였다.

3. 하드디스크 및 플래시 기반의 디스크에서 내부병렬성을 통하여 프리페치 시간을 단축시키는 스케쥴링 기법을 제안하였다. 특히, 플래시 기반의 디스크에서 다양한 병합 및 명령 큐잉을 이용하여 내부병렬성을 높임으로써 프리페치 시간을 단축시키는 기법을 제안하였다.

4. 내부병렬화를 통한 프리페치 속도 향상, 그리고 대상 프로그램과 프리페칭 작업의 병렬적 수행은 기동시간에 큰 영향을 준다. 하지만 이 둘은 이율배반 (trade-off) 관계에 있기 때문에, 본 논문에서는 디스크의 입출력 패턴, 디스크 장치의 특성, 운영체제의 파일시스템 및 블록 계층의 특성 등을 종합적으로 고려하여 기동시퀀스의 프리페칭을 빠르게 하는 동시에 프로세서와 디스크 프리페칭 처리의 병렬성을 높여 응용프로그램의 기동을 빠르게 하는 기법을 설계, 구현, 그리고 평가하였다.

1.3 논문의 구성

본 논문의 구성은 다음과 같다. 제 2장에서는 하드디스크와 SSD의 특성 및 리눅스의 디스크 입출력 부 시스템의 특징을 살펴보고, 응용프로그램의 기동시간을 단축시키기 위한 기존 연구를 살펴본다. 제 3장에서는 응용프로그램 기동 시의 디스크 입출력 및 동작 특성을 분석하고, 제 4장에서 커널 수준의 프리페칭 프레임워크의 설계 및 구현에 대해서 소개하고 데스크탑 및 모바일 워크로드를 이용하여 성능을 평가한다. 제 5장에서는 유저 수준의 프리페쳐 구현 방법에 대해서 소개하고, 유저 수준의 실행시간 프리페쳐의 성능 평가 및 결과 분석이 제시된다. 마지막으로 제 6장에서는 본 연구 결과를 요약하고 향후 연구 방향에 대해 논의한다.

제 2 장　연구 배경

2.1 범용 디스크 드라이브

PC, 노트북, 스마트패드, 그리고 스마트폰 등 다양한 크기와 성능의 개인용 기기는 폼팩터 제한 및 사용자의 요구 사항 등에 따라 하드디스크나 플래시 기반의 저장 장치를 장착한다. 보조 저장 장치는 응용프로그램 기동 시간에 가장 큰 영향을 주는 하드웨어 구성요소 중 하나로 사용자 및 시장의 요구사항을 만족시키기 위하여 끊임없이 진화해 왔다. 이번 절에서는 범용 디스크 드라이브의 최신 경향 및 특성에 대해서 살펴본다.

2.1.1 하드디스크 드라이브

하드디스크 드라이브는 순차 처리 성능이 높고 용량 대비 비용이 낮아 보조기억장치로 많이 사용되어 왔다. 또, 1961년 등장 이후 집적도 및 성능을 개선하기 위하여 끊임없이 진화해왔다. 예를 들면, 2004년 도시바에 의해 실용화된 수직자기 기록방식 (perpendicular magnetic recording, PMR)은 기존의 수평자기 기록방식 (longitudinal magnetic recording, LMR) 대비 100%의 집적도 향상을 가져왔다. 또, 분당회전수 (revolutions per minute, RPM), 기록 기술, 내장 캐시 크기, 탐색 및 정착 (settle) 시간 등의 개선을 통하여 매년 평균 40%의 처리율 향상을 달성해 왔다 [63]. 분당회전수는 하드디스크의 에너지 소모에 큰 영향을 주는데 입출력 요청에 따라 동적으로 분당회전수를 조정하여 성능은 유지하면서 에너지소모는 줄이는 기법이 연구 및 제품화되었다 [63]. 하드디스크의 집적도는 인치당 트랙 수와 트랙당 섹터 수의 곱으로 결정된다. 최근에는 하드디스크의 제조결

함을 숨기기 위하여 이 두 값을 플래터의 상태에 따라 동적으로 선택하는 적응형 존 (adaptive zoning) 기법이 많은 하드디스크에 적용되었고 이는 같은 모델의 하드디스크라고 하여도 성능이 다를 수 있음을 의미한다 [49].

이러한 많은 노력들에도 불구하고 하드디스크는 시스템 병목 (bottleneck) 현상의 가장 큰 원인으로 여겨져 왔다. 이는 하드디스크에서 데이터를 읽기 위해서는 데이터가 있는 위치로 디스크 헤드를 이동시키는 기계적인 동작을 필요하기 때문이다. 디스크의 접근 시간은 탐색 시간과 회전 지연 시간으로 구성되며 성능 개선이 굉장히 느린 것이 특징이다. 초당 처리되는 입출력 수 (I/O per second, iops)를 향상시키기 위한 방법으로 네이티브 명령 큐잉 (native command queuing, NCQ) 기법이 활용되고 있다 [48]. NCQ는 데스크탑이나 노트북에 많이 사용되는 호스트 인터페이스인 SATA-2 표준으로 등장한 기법이다. NCQ는 최대 32개의 입출력 요청을 받아 디스크 내부적으로 접근 시간을 최적화시키기 위하여 처리 순서를 바꾼다. 또, 인터럽트를 각각의 요청에 대해서 발생시키지 않고 짧은 시간 내에 완료된 여러 개의 요청에 대해서 한 번만 인터럽트를 발생시킨다. I/O 요청을 요청 큐에 충분히 입력한 경우 초당 처리되는 입출력 수가 약 50% 개선되지만 처리 순서를 바꾸기 때문에 대부분의 제품에서 기아현상이 발생하기도 한다. 이를 방지하기 위해서 한 요청이 다른 요청들 때문에 순서가 미루어지는 최대 횟수를 제한하는 연구가 있다 [32].

2.1.2 NAND 플래시 기반 Solid-State Drive (SSD)

플래시 메모리 기반의 SSD는 기계적인 움직임이 없어서 접근 시간이 거의 일정하고 하드디스크에서 수 ms가 걸리던 접근 시간을 수 십 us로 단

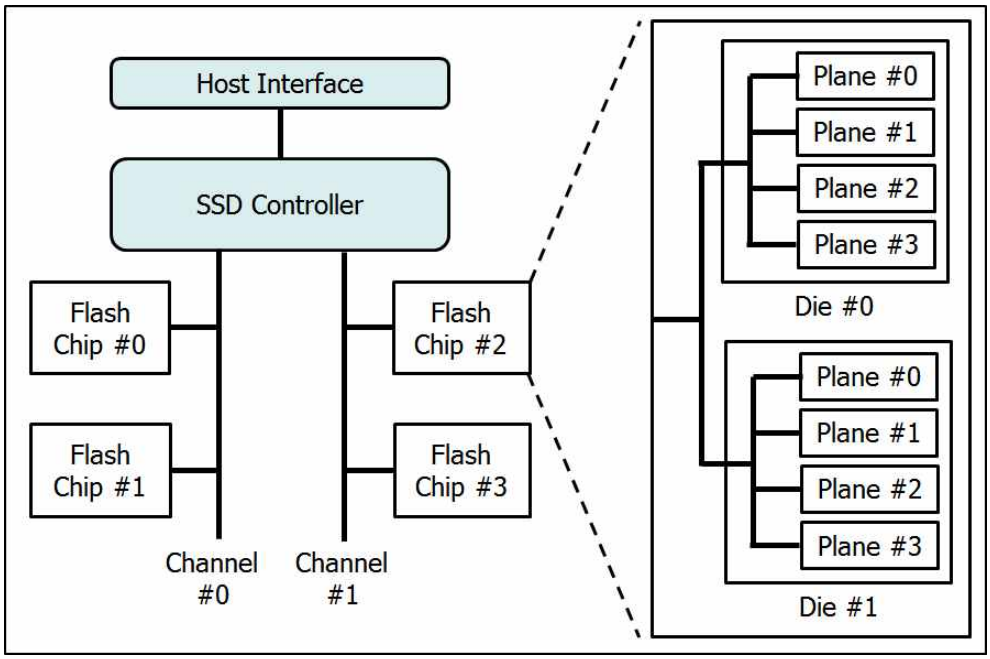

그림 1 SSD의 내부구조

축시켰다. 이 때문에 임의 접근 패턴에서 하드디스크 보다 수십에서 수백 배 빠른 성능을 보여준다. SSD는 일반적으로 다수의 NAND 플래시 칩들로 구성되고 이 칩들을 얼마나 병렬적으로 이용하는지에 따라 처리율이 크게 달라진다 [53]. 또, 단일 칩 내부에서도 다이 (die), 플레인 (plane) 수준에서 병렬적 처리를 지원한다.

NCQ가 지원되는 경우, 여러 개의 입출력 요청을 동시에 명령큐에 요청함으로써, 여러 개의 작은 요청들이 SSD의 병렬적 사용을 유도하여 처리율을 높일 수 있다. 또, 입출력 요청 크기가 큰 경우에는 하나의 명령만 처리되는 경우에도 병렬성을 이용할 수 있다. 이는 현재 판매되는 대부분의 제품에 장착된 FTL (flash translation layer)에서 페이지 수준의 사상(mapping)을 이용하고 큰 쓰기 요청을 여러 플래시 칩에 나누어 저장하기 때문이다. SATA 뿐 아니라 스마트폰이나 스마트패드와 같은 모바일용 차

세대 인터페이스인 UFS (universial flash storage) 표준에서 명령 큐잉을 지원한다 [68]. 또, 대부분의 데스크탑 PC 용 SSD 제품은 NCQ를 지원하지만 SD메모리는 커맨드 큐잉을 공식적으로 지원하지 않는다. 명령큐잉이 지원되지 않는 경우에는 인접한 요청을 병합하여 큰 요청으로 처리함으로써 내부병렬성을 이용할 수 있다. 에너지 소비와 폼팩터에 제한이 크고 큰 공간이 필요하지 않은 스마트폰에서는 플래시 기반의 저장장치가 선택이 아닌 필수가 되었다.

여전히 SSD의 비트 당 비용은 하드디스크보다 10배 정도 높다. 하지만 셀 당 비트 수를 늘리고 제조공정의 혁신을 통하여 셀 (cell)의 단위 크기 (feature size)를 줄임으로써 SSD의 가격하락이 가속화되어 왔다. 하지만 셀 당 비트 수를 늘리거나 셀의 단위 크기가 줄어들면 플래시 칩의 접근 시간, 에러율, 프로그램 횟수 등의 특성이 오히려 악화된다 [29]. 또, 높은 에러율을 극복하기 위하여 더 강력한 순방향 에러복구코드 (forward error correction, FEC)를 이용하고 플래시의 수명을 늘리기 위해 압축이나 중복 데이터제거 기술과 같은 많은 처리를 SSD 컨트롤러가 수행해야 한다. 이러한 복잡한 처리로 인하여 입출력 지연시간은 악화될 수 있다.

2.1.3 하이브리드 하드디스크

개인용 기기를 사용할 때 발생되는 디스크 접근 형태는 응용프로그램 기동이나 파일 탐색 작업과 같이 임의 접근 패턴을 보이는 경우, 그리고 대용량 미디어 파일을 재생하거나 큰 파일을 복사할 때 발생하는 순차 접근 패턴을 보이는 경우가 큰 비중을 차지한다. 일반적으로 임의 접근 패턴에서 요청되는 데이터의 크기는 작고 순차 접근 패턴에서의 데이터의 요청 크기

는 크다. 이와 같은 특성을 이용하여 하드디스크 수준의 비용으로, 임의 접근 속도는 플래시 장치와 비슷하고 순차 접근 속도는 하드디스크 수준이 되도록 하는 저장장치가 하이브리드 하드디스크이다. 하이브리드 하드디스크는 작은 용량의 플래시 메모리를 하드디스크의 캐시로 이용하며 작은 임의 접근 패턴을 보이는 데이터는 플래시에 저장하고 큰 순차 접근 패턴을 보이는 데이터는 하드디스크에 저장하여 적은 비용으로 높은 처리율과 짧은 응답시간을 제공한다.

2007년에 삼성전자는 80~200GB의 하드디스크에 128~256MB의 NAND 플래시를 통합한 2.5인치 하이브리드 하드디스크를 출시했다. 이 모델은 ATA8에서 명세한 NVCACHE 관련 ATA 커맨드를 이용하여 하드디스크의 내용을 플래시에 적재하고, 어떤 디스크 블록을 캐싱할 지에 대한 처리는 윈도우즈 운영체제에서 제공하는 레디드라이브 (ready drive) 모듈이 수행하였다. 같은 해에 시게이트도 160GB의 하드디스크에 256MB의 플래시를 통합한 모멘터스 5400PSD를 출시하였다. 하지만 이러한 제품들은 작은 캐시 사이즈로 인하여 큰 효과가 없어, 시장에서 사장되었다. 2010년에 시게이트는 250~500GB의 하드디스크에 4GB의 큰 플래시 메모리를 장착한 모멘터스 XT 모델을 출시하였고, 디스크 내부에 어느 부분을 캐싱할 지에 대한 처리 모듈이 포함되어 있어서 운영체제에서 캐싱 기능을 지원여부에 관계없이 독립적으로 동작한다. 또, 플래시 캐시의 크기가 크기 때문에 임의 접근 패턴으로 요청되는 블록들을 효과적으로 캐싱할 수 있다. 이 제품은 NVCACHE 관련 ATA 커맨드를 지원하지 않기 때문에 운영체제에서 하드디스크의 어느 부분이 캐싱되었는지 알 수가 없고 특정 블록을 캐싱하도록 요청할 수도 없다. 최근 시게이트는 500~750GB의 하드디스크에 8GB의 플래시 메모리를 장착한 모멘터스 XT 2세대를 출시하였다.

2.2 리눅스의 디스크 입출력 부 시스템

이번 절에서는 리눅스의 디스크 입출력 계층과 디스크캐시에 대해서 설명한다. 또, 디스크캐시 무효화 방법과 무효화의 불완전성, 그리고 입출력 요청의 병합 방법인 입출력 플러그/언플러그에 대해서 알아본다.

2.2.1 리눅스의 디스크 입출력 스택

그림 2는 리눅스에서 입출력 처리 과정을 나타내는 다이어그램이다. 프로세스는 유저 수준의 코드이므로 직접적으로 장치에 접근하는 것은 제한된다. 따라서 프로세스가 디스크 입출력을 요청하는 방법은 디스크 입출력

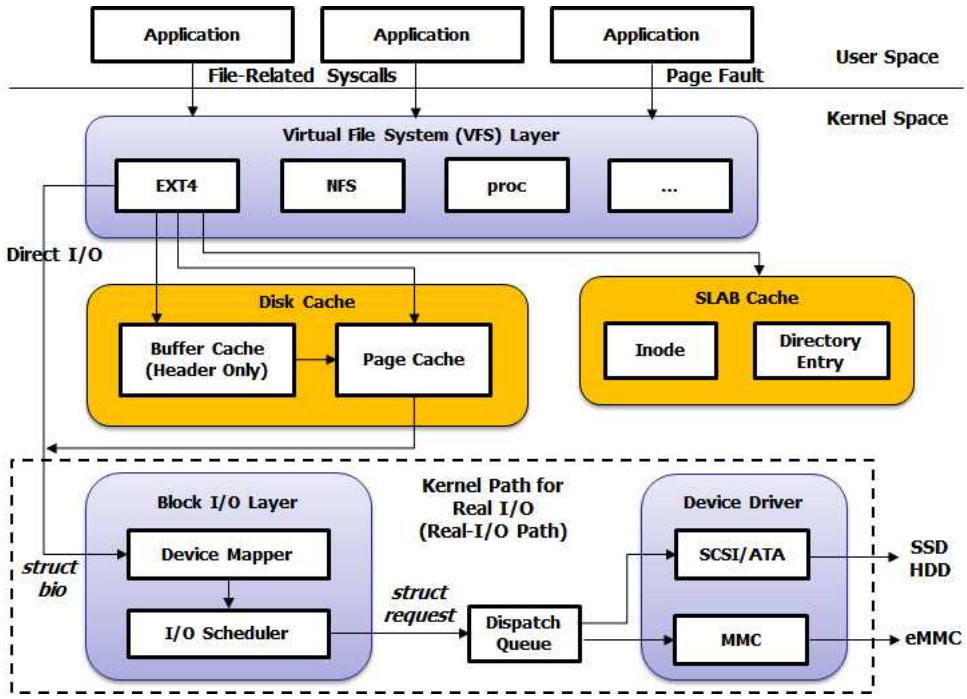

그림 2 리눅스의 디스크 입출력 처리 흐름

과 관련된 시스템 콜 (system call)을 호출하거나, 디스크 영역이 메모리 주소 공간에 사상된 (memory-mapped) 가상주소에서 페이지 폴트 (page fault)가 발생한 경우로 나눌 수 있다. 이 두 가지 경우에 디스크 입출력 요청을 하기 위하여 가상파일시스템 (virtual file system, VFS) 계층을 거친다. 가상파일시스템은 실제 파일시스템을 일관된 자료구조와 인터페이스를 통하여 처리될 수 있도록 추상화시킨 계층이다. 따라서 ext4, ntfs, btrfs 등과 같은 실제 파일시스템은 추상화된 자료구조와 인터페이스 상에서 처리될 수 있도록 구현되어야 한다. 리눅스의 VFS 구조는 ext 계열의 파일시스템에 최적화되어 있어서 이들 파일시스템에서 추상화에 필요한 변환 오버헤드가 가장 적다.

파일시스템은 저마다 물리적으로 데이터를 관리하는 방식 (file system format)을 정의한다. 입출력 요청 시에 파일시스템은 요청된 블록이 무엇인지 분석하여 이를 처리하기 위해 추가적으로 읽히거나 쓰여져야 하는 메타데이터 블록들이 있는지 조사하고, 이러한 블록들이 있다면 이들을 모두 읽거나 쓴 후에 요청된 데이터를 처리할 수 있다. 파일시스템은 실제 디스크 입출력을 발생시키기 전에 원하는 블록이 디스크캐시에 존재하는지 검사한 후, 디스크캐시에 없는 경우에 디스크 입출력을 위한 요청을 수행한다. 예외적으로 파일을 열 때, 플래그 인자에 O_DIRECT를 포함시킨 경우에는 디스크캐시를 거치지 않고 직접 입출력을 시도한다. 리눅스 커널의 bio (block I/O) 구조체는 논리블록번호 (logical block number, LBN) 상 연속된 영역의 입출력 요청을 나타내는데 사용되는 구조체로 실제 디스크 입출력을 처리하기 위해서 장치 사상자 (device mapper) 계층을 거쳐 입출력 스케쥴러로 전달된다. 장치 사상자 계층은 물리적인 파티션을 논리적으로 구성하는 논리볼륨관리자 (logical volume manager, LVM) 기능을 수행하거나 여러 디스크 장치를 소프트웨어 RAID로 처리하는 역할을 한다. 입출

력 스케쥴러는 전달된 요청들의 순서를 바꾸어 디스크의 처리율을 향상시키고, 요청들 간에 공평성(fairness)을 개선하거나, 아직 처리되지 않은 bio 요청과 새로 요청을 시도한 bio가 논리블록번호 상 인접한 경우 하나의 request로 통합하여 처리의 효율성을 높이는 역할을 한다. 리눅스 커널의 request 구조체는 논리블록번호 상 연속적인 하나의 디스크 요청을 표현할 수 있으며 디바이스 드라이버와 디바이스 간의 처리 단위이기도 하다.

2.2.2 리눅스의 디스크캐시

리눅스 커널 버전 2.2 에서는 블록 장치의 캐싱을 위하여 버퍼캐시(buffer cache)가 사용되었다. 각 버퍼는 1KB의 데이터 저장 공간과 관리 헤더 기록을 위한 버퍼 헤더를 필요로 하였다. 또, 버퍼캐시는 디바이스 ID와 블록 번호를 헤더에 저장하고 이 두 값을 이용하여 해싱을 하였다. 빠른 검색을 위하여 장착된 메인 메모리 크기에 따라 해시테이블의 크기가 동적으로 결정되었다.

2000년대에 들어서면서 버퍼캐시와 같이 물리적인 위치 정보를 통하여 디스크캐싱을 하던 많은 운영체제에서 파일 수준의 캐싱을 사용하기 시작하였다. 파일 수준의 캐싱은 기존의 디바이스 번호와 디바이스 내 블록 번호를 통하여 캐싱하는 것이 아닌 파일의 ID와 파일 내 블록 번호 (또는 페이지 번호)로 캐싱하는 것을 의미한다. 리눅스에서 파일 수준의 캐싱은 파일의 아이노드 (inode) 번호와 파일 내 페이지 번호를 통하여 캐시 탐색이 수행되고, 캐시의 빠른 탐색을 위해서 기수 트리 (radix tree) 알고리즘을 이용한다. 파일 수준의 캐싱은 파일 수준의 순차 접근 경향을 쉽게 파악할 수 있고, 파일의 아이노드 별로 검색 트리를 따로 관리하기 때문에 버퍼캐시와

같이 전체 블록장치를 대상으로 관리하는 경우에 비해 검색 대상이 크게 줄어든다는 장점이 있다.

리눅스 커널 2.6에서 버퍼캐시와 페이지캐시가 통합되었다. 버퍼캐시는 버퍼헤더만 존재하고 데이터는 해당 디바이스의 아이노드를 통하여 페이지 캐시 처리가 이루어진다. 이렇게 인터페이스를 통합하지 않은 이유는 파일시스템의 블록크기와 메모리 페이지의 크기가 다를 수 있기 때문이다. 예를 들어, 1KB의 파일시스템 블록 크기를 갖는 ext3 파일시스템으로 포맷된 파티션에서 접근의 공간지역성이 낮은 아이노드 테이블의 블록을 페이지 크기보다 작은 블록 크기로 읽는 것이 유리할 수 있기 때문이다. 파일시스템 상의 블록 크기가 1KB인 경우, 하나의 페이지는 4개의 버퍼 헤더로 구성될 수 있다. 또, 4KB를 담을 수 있는 페이지캐시 한 엔트리에 1KB의 정보만이 유효할 수도 있다. 정규 파일의 경우 페이지캐시를 통하여 캐싱하므로 파일시스템 블록 크기가 아닌 페이지 크기 단위로 입출력 및 캐싱 처리가 수행된다.

디스크 블록의 캐싱은 여러 수준에서 수행될 수 있다. 프로세스 가 할당한 메모리에 캐싱을 할 수도 있고, 표준 입출력 함수를 사용해서 라이브러리 수준에서 캐싱을 수행할 수도 있다. 또, 커널 수준에서 디스크캐시가 버퍼캐시와 페이지캐시만 있는 것은 아니다. 슬랩 (slab) 또는 slob, slub과 같은 slab의 변종 오브젝트 할당자가 있다 [57][61]. slab은 자주 쓰는 작은 오브젝트들을 반복적으로 할당 및 해제할 때 발생하는 페이지의 내부 단편화를 줄여서 메모리를 효율적으로 사용하고자 함이 목적이다. 또, 해싱을 통하여 이 오브젝트들의 검색을 빠르게 할 수도 있다. 디스크 블록과 관련된 자주 사용되는 오브젝트는 inode와 dentry가 있는데 이들은 slab을 통하여 이중으로 캐싱된다. inode나 dentry는 파일시스템 블록 크기보다 훨

씬 작기 때문에 해당 디스크 블록의 일부 오브젝트만 slab에 캐싱될 수 있다.

리눅스 커널은 디스크캐시를 무효화시키는 커널 함수와 프록 (proc) 파일을 제공한다. 후자의 방법은 리눅스 터미널 상에서 /proc/sys/vm/drop_caches 파일에 숫자 값을 써 넣음으로써 가능하다. 1을 쓰면 페이지캐시를 무효화시키고 (invalidation), 2를 쓰면 inode와 dentry 할당자의 캐싱된 오브젝트들를 무효화시킨다. 마지막으로 3을 쓰게 되면 페이지캐시, inode slab 및 dentry slab 캐시를 모두 무효화한다. 중요한 점은 페이지캐시와 버퍼캐시의 경우 락 (lock)이 걸려있지 않은 경우 거의 완벽하게 무효화되지만 slab 캐시의 경우 완벽히 무효화시키게 되면 전체 시스템의 성능이 급격히 감소하기 때문에 완벽히 무효화하지 않는다.

2.2.3 입출력 스케쥴러의 종류 및 특징

리눅스에서 많이 사용되는 디스크 입출력 스케쥴러로는 noop, deadline, anticipatory, cfq가 있다. 하지만 anticipatory 스케쥴러는 비작업보장성 (non-work-conserving) 때문에 리눅스 커널 버전 2.6.33부터 메인라인 커널에서 제거되었다. 또, SSD와 같이 접근 시간이 거의 일정한 장치를 위한 공평성 보장 입출력 스케쥴러로 fiops (fair in out performance scheduler) 가 있다 [69].

noop (no-operation) 스케쥴러는 아무일도 하지 않는 가장 간단한 스케쥴러이다. SSD나 메모리 기반 저장장치와 같이 입출력 순서의 변경을 통하여 성능향상을 보기 어려운 장치에 적합하다.

데드라인 (deadline) 스케쥴러는 읽기나 쓰기 요청을 논리블록번호로 정렬된 큐와 선입선출 (first-in first-out, FIFO) 큐에 입력하고 각 요청의 처리에 데드라인을 설정한다. FIFO 큐에서 데드라인을 넘긴 요청이 없는 경우 논리블록번호로 정렬된 큐에 삽입된 요청을 처리하는 방식이다. 이 방식은 데드라인을 통해 요청의 기아현상 (starvation)을 방지하고 데드라인을 넘기지 않는 한도 내에서 논리블록정렬을 통하여 하드디스크 같은 장치에서 디스크의 헤드 움직임을 최소화한다. 또, 일반적으로 쓰기보다 읽기의 데드라인을 짧게 설정하는데 이는 읽기 요청은 프로세스를 블록 (block)시켜 프로세서의 활용도를 낮추고 결국 디스크 요청도 줄어들 수 있는 결과를 초래 수 있기 때문이다.

예측 (anticipatory) 스케쥴러는 데드라인 스케쥴러와 유사하지만 다른 점은 읽기 요청을 처리한 후 정해진 짧은 시간 (기본적으로 6ms) 동안 인접한 요청을 기다리는 점이다. 이는 프로세스의 디스크 요청에 대한 처리가 완료된 후 다음 블록을 요청하기 위해서 짧은 입출력 스택 처리 시간이 필요한데, 요청하려고 하는 블록이 최근에 처리한 블록과 가까운 경우 이것을 처리하고 다른 요청을 처리하면 디스크 헤드 움직임을 크게 줄일 수 있다. 하지만 비작업보장성으로 인해 워크로드의 특성에 따라 성능 향상이 큰 경우도 있지만 반대로 성능 저하가 클 때도 있다.

cfq (completely fair queuing) 스케쥴러는 기본적으로 논리블록번호로 정렬된 순서로 입출력 요청을 처리하는데, 큐를 하나만 두는 것이 아니라 프로세스나 프로세스 그룹 별로 요청 큐를 따로 두고 각 요청 큐에 대한 입출력 시간을 공평하게 배분하는 방식을 통하여 공정성을 높인다. 또, SSD와 같은 장치에서는 논리블록번호로 입출력 요청을 정렬하는 것은 크게 효과가 없기 때문에 SSD와 같은 장치를 위해 초당 입출력 수에 대한 공평성

을 보장하면서 입출력 정렬은 수행 하지 않는 fiops 스케쥴러가 있다.

2.2.4 입출력 플러그/언플러그

리눅스는 오래전부터 플러깅 (plugging) 기법을 블록 디바이스에 적용해 왔다. 플러깅은 디스크가 유휴 (idle) 상태일 때 입출력 요청이 발생한 경우, 요청을 즉시 처리하지 않고 짧은 시간동안 기다림으로써 인접한 요청이 병합될 수 있는 시간적 기회를 주고, 이를 통하여 디스크의 처리 효율을 높이는 기법이다. 예를 들면, ext3 파일시스템의 경우 아이노드 내의 직접 블록 포인터로 12개의 블록을 가리킬 수 있는데 13개의 블록 크기를 갖는 파일이 있다면 데이터 블록 영역은 12개의 앞 데이터 블록들, 단일 간접 포인터 블록, 13번째 데이터 블록 순서로 인접하게 할당되는 경향이 있다. 이 경우에 어떤 프로세스가 파일의 전체 부분을 읽는 요청을 했다면 첫 12개의 블록 요청과 단일 간접 포인터 블록은 플러깅에 의해서 하나의 요청으로 처리된다. 13번째 블록은 단일 간접 포인터 블록이 읽힌 후에 처리될 수 있으므로 이전 요청이 완료되기 전 까지는 요청될 수 없다.

리눅스 커널 2.6.39 이전 버전에서는 플러깅을 전체시스템 (global) 수준에서 또는 디바이스 별로 관리가 되었지만 커널 2.6.39부터는 프로세스나 프로세스 그룹별로 관리가 된다. 또 한 가지 중요한 변화는 플러깅 기법이 비작업보장성을 유발하므로 리눅스 커널 2.6.39부터는 이를 커널에서 자동으로 처리하지 않는다. 대신 개발자가 필요에 따라 명시적으로 플러그/언플러그를 처리할 수 있도록 인터페이스를 제공하고 있다.

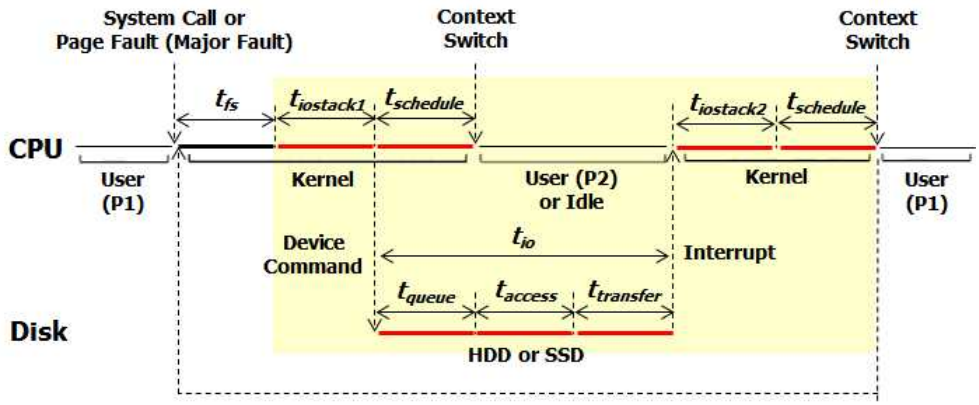

그림 3 디스크 요청에 대한 처리 시간 요소

2.2.5 프리페치 성공 시 절약되는 프로세서 시간 분석

성공적인 프리페칭은 디스크의 시간 뿐 아니라 프로세서 시간도 일부 단축시켜 준다. 그림 3은 프로세스의 디스크 입출력 요청이 수행되는 과정을 커널 내부의 작업 내용에 따라 구분한 것을 보여준다. 프리페칭은 읽기 처리와 관련이 있으므로 읽기와 연관지어 설명하도록 하겠다. 유저 프로세스는 페이지 폴트나 시스템콜을 통해서 디스크 서비스를 받을 수 있다. 이 때 커널의 파일시스템 코드를 통해서 원하는 블록의 위치를 계산하고 해당 블록이 디스크캐시에 있는지 검색한다. t_{fs}가 이러한 처리에 소비되는 시간에 해당한다. 성공적인 프리페칭을 통하여 필요한 블록을 디스크캐시에 미리 적재해 놓았다면 캐싱된 내용을 전달해 줌으로써 모든 처리가 끝난다.

디스크캐시에서 적중 (cache hit)이 되지 않은 경우 실제 디스크 입출력을 요청하기 위해서 블록 계층, 입출력 스케쥴러, 디바이스 드라이버를 수행하게 되는데 $t_{iostack1}$이 이 부분을 수행하는데 소요되는 시간을 의미한다. 디바이스 드라이버에서 디스크 하드웨어에 입출력을 요청한 후에, 입출력을 요청한 프로세스를 대기상태 (wait)로 변경하고, 실행가능 (runnable)

상태에 있는 다른 프로세스에 프로세서를 사용권을 넘겨준다. 그림 3에서 알 수 있듯이 스케쥴링은 디스크 하드웨어의 내부 동작과 동시에 수행될 수 있다. 요청된 데이터의 준비가 완료되면 디스크는 인터럽트를 발생시킨다. 이 때, 데이터가 준비되기를 기다리고 있는 프로세스들을 깨우기 위해 프로세스의 상태를 준비상태 (ready)로 변경한다. 이 작업에 드는 시간은 $t_{iostack2}$이다. 마지막으로 스케쥴러에 의해 디스크 서비스 완료를 기다리고 있는 프로세스로 스케쥴링되는데 필요한 $t_{schedule}$의 시간이 소비된다.

결과적으로, 성공적인 프리페칭은 그림 3에서 큰 사각형으로 표시된 부분을 처리하는데 소비되는 시간을 줄여준다. 여기에는 디스크 시간 뿐 아니라 디스크캐시 탐색실패 (cache miss)로 인해 실제 디스크 입출력 처리를 하고 동기화를 수행하기 위한 프로세서 사용 시간도 포함되어 있다. 디스크의 동작과 동시에 수행될 수 있는 프로세서 처리 부분을 제외하면, 성공적인 프리페칭을 통하여 줄어드는 프로세서 시간 $t_{reducible}$은 다음과 같이 표현될 수 있다.

$$t_{reducible} = t_{iostack1} + t_{iostack2} + t_{schedule}$$

2.3 응용프로그램의 빠른 기동을 위한 기존 연구

현재까지 대부분의 연구는 하드디스크의 낮은 성능을 소프트웨어 및 하드웨어 수준에서 개선하는 기법에 관한 것이다. 이번 절에서는 이러한 기법들을 응용프로그램의 기동속도 단축에 적용한 기존 연구들에 대해서 알아본다.

2.3.1 응용프로그램의 빠른 기동을 위한 디스크캐싱 기법

윈도우즈 운영체제는 시스템 부팅 및 프로그램 기동시간의 대부분을 차지하는 하드디스크 헤드의 이동시간을 개선하기 위해 윈도우즈 프리페쳐(Windows prefetcher)를 운영체제에 탑재하였다 [55]. 시스템이 처음 부팅되거나 응용프로그램이 기동될 때, 메모리 관리자는 접근되는 코드나 데이터의 정보를 윈도우즈 프리페쳐에 통보하고, 프리페쳐는 이를 파일시스템 수준의 정보로 변환하여 저장한다. 이 후에 같은 프로그램이 실행될 때에는 변환된 정보를 통하여 최적화된 순서로 기동시퀀스를 프리페칭 한 후 해당 프로그램을 수행시킴으로써 기동시간을 단축시킨다.

추가적인 성능향상을 위하여 모든 기동시퀀스를 기본적으로 3일마다 연속된 충분히 큰 빈 공간으로 옮긴다. Mac OS의 경우, 부팅시간을 단축시키기 위하여 부트캐시 (bootcache) 기법을 탑재하여, 부팅 시 접근되는 많은 블록들을 논리블록번호로 정렬된 순서대로 읽어 들인다 [58]. 이 밖에도 윈도우즈는 사용자의 프로그램 사용패턴을 분석하여 가까운 시간 내에 수행될 가능성이 높은 프로그램의 기동시퀀스를 프리페칭하는 슈퍼페치(superfetch)를 제공한다 [55].

구글-프리페치 [16]는 윈도우즈 프리페쳐와 비슷한 방식으로 작동한다. 리눅스는 물리 메모리 페이지마다 작은 크기의 페이지 구조체를 할당하여 페이지의 속성을 관리한다. 구글-프리페치는 응용프로그램의 기동시퀀스 수집을 위해 처음 기동될 때, 시스템의 모든 페이지 구조체의 참조 속성을 제거한다. 그리고 프로그램을 기동 시킨 후 10초 이후에 참조 속성이 켜진 페이지와 페이지캐시로부터 방출된 블록 정보를 이용하여 프로그램 기동 중에 접근된 블록의 정보를 얻어내고 디바이스 번호, 아이노드 번호, 파일 내 옵셋 번호로 정렬하여 프리페치 정보 파일에 저장한다. 이 후에 같은 프로그램을 기동하면 저장된 프리페치 정보 파일에 기록된 순서대로 기동 관련 블록들을 프리페칭하여 디스크 시간을 크게 줄이고 프로그램을 수행함으로써 기동시간을 단축시킨다.

구글-프리페치에서 사용하는 기동시퀀스 수집 방법은 디스크캐시를 무효화시키지 않은 상태에서 추출하기 때문에 slab을 통하여 이미 캐싱되어 있는 아이노드나 디렉토리 엔트리와 관련된 블록은 수집되지 않을 수 있다. 또, 파일 수준에서 정렬된 기동시퀀스의 프리페칭은 상황에 따라 많은 디스크 헤드의 움직임을 유발시킬 수 있다. 리눅스 프리페치는 윈도우즈 프리페치와 마찬가지로 부팅을 위한 프리페치를 지원하고 응용프로그램의 기동시퀀스를 인접한 디스크 공간에 배치하는 기능을 지원한다. 구글-프리페치의 조각모음 (defragmentation) 기능은 ext3 호환 파일시스템 용으로 제작되었는데 ext3 파일시스템처럼 아이노드 번호에 해당하는 디스크 블록의 위치가 정해져 있는 경우에는 아이노드를 옮기면 아이노드 번호가 바뀌기 때문에 이 값을 저장하고 있는 디렉토리 엔트리를 수정해야한다. 특히, 해당 아이노드의 하드링크 계수가 1보다 큰 경우에는 아이노드를 사용하는 모든 디렉토리 엔트리를 찾아서 변경할 수 있어야만 이동이 가능하다.

Prelinking은 동적 라이브러리가 적재될 주소를 미리 지정함으로써 재배치로 인한 오버헤드를 줄인다. 동적 라이브러리가 제공하는 재배치의 장점을 포기하는 대신 이로 인한 계산 시간을 줄여서 응용프로그램 기동시간이 단축된다. Prelinking을 위한 도구로 리눅스는 prelink [67], 그리고 Mac OS는 prebinding [70]을 제공한다.

Preloading은 프로그램이 사용하는 동적 라이브러리들을 유휴시간에 미리 읽어 놓음으로써 응용프로그램의 기동 시에 동적 라이브러리 파일을 디스크로부터 읽는 시간을 줄여 기동시간을 단축하는 기법이다. Preloading을 위한 유틸리티로 리눅스에 preload가 제공된다 [6]. preload는 유저모드의 데몬 프로세스로 동작하며 20초마다 깨어나 어떠한 프로세스가 실행되고 있는지 proc 파일시스템을 통하여 감시하고, 각 프로세스가 어떠한 실행파일과 라이브러리를 메모리 공간에 사상하였는지를 /proc/<프로세스 ID>/maps 파일을 통하여 수집한다. 이를 바탕으로 사용자의 프로그램 사용 패턴을 분석하고 가용 메모리 크기를 고려하여 가까운 시간 내에 수행될 가능성이 있는 프로그램의 실행파일과 라이브러리를 디스크캐시에 미리 적재한다. Preloading 기법은 메모리 사상된 파일만을 프리페칭하기 때문에 블록 캐싱의 재현정확도가 낮고 블록 단위가 아닌 파일 단위로 프리페칭을 수행하기 때문에 기동시퀀스가 아닌 블록이 디스크캐시에 적재되는 문제가 있다. 이는 메모리 크기가 작은 장치에서 심각한 메모리 낭비를 유발할 수 있다. Jung [66] 등은 임베디드 기기에서 사용되는 프로그램에서 prelinking과 preloading을 통한 기동시간 단축 정도를 분석 및 평가하였다.

Yan [34] 등은 윈도우즈 폰에서 사용자의 응용프로그램 사용 패턴을 시간, 장소, 프로그램 연관성 등을 이용하여 분석하고, 미리 실행 시 소비되

는 에너지와 단축되는 기동시간 등을 고려하여 프로그램을 미리 실행시킴으로써 기동시간을 단축하는 기법을 제안하였다. 기동시퀀스의 프리페칭뿐 아니라 기동 시 사용되는 네트워크 데이터 등도 미리 요청함으로써 기동시간을 기존 방식보다 단축하였다. 하지만 디스크 블록 이외에도 더 많은 종류의 데이터를 미리 처리하는 만큼 프로그램 사용 패턴 분석을 통한 예측의 정확도가 중요하다.

2.3.2 범용 워크로드의 빠른 응답을 위한 디스크캐싱 기법

프리페칭의 대상을 응용프로그램 기동시퀀스로 한정하지 않은 범용 워크로드 프리페칭 기법들은 1) 항상 블록 접근을 감시해야하고, 2) 블록들 간의 복잡한 상관관계를 분석해야 하며, 3) 프리페칭된 블록들이 실제로 읽히는지 또는 얼마나 많은 블록을 프리페칭해야 하는지에 대한 지속적인 분석 오버헤드가 요구된다. 범용 프리페칭 기법은 응용프로그램의 빠른 기동에도 적용이 가능하지만 일반적으로 오버헤드가 크기 때문에 실제로 운영체제에 탑재되기는 어렵다.

블록 수준에서 접근 패턴을 분석하고 분석된 내용을 바탕으로 이후의 접근 블록을 예측하는 방식은 범용 프리페칭에서 가장 널리 이용되는 방법 중 하나이다. C-Miner [24]는 접근되는 블록 시퀀스에서 자주 발생하는 요청 패턴을 블록 수준에서 분석한다. 블록 a와 b의 요청 사이에 다른 요청의 개수가 임계값보다 작으면 이 둘은 상관관계가 있다고 판단하여 관계 정보를 관리한다. 또, C-Miner는 블록 a가 요청된 후 블록 b가 요청될 확률과 횟수를 고려한다. 즉, a가 요청된 후 b가 요청될 확률이 적거나, 확률은 높아도 너무 드물게 발생하는 패턴은 상관관계 신뢰도가 낮다고 판단하여 상

관관계 정보를 유지하지 않는다. 이는 정보 관리 및 판단실패로 인한 오버헤드가 성공적인 프리페칭으로부터 얻는 이득보다 클 수 있기 때문이다. 라이브러리 블록은 여러 응용프로그램의 기동시퀀스에 공통적으로 나타날 수 있기 때문에, C-Miner의 상관관계 분석 방법을 사용할 경우, 라이브러리 블록들은 상관관계가 정보가 형성되지 않을 가능성이 높다. 또, 드물게 실행되는 프로그램은 상관관계 신뢰도가 낮아 프리페칭되지 않을 수 있다.

DiskSeen [62]은 논리블록번호 상 가까운 위치에 있는 블록들 간의 접근 상관관계를 분석하여 프리페칭에 이용하는 기법이다. 파일 수준의 접근 패턴 분석은 하드디스크의 물리적인 블록 배치 정보를 활용할 수 없기 때문에 DiskSeen은 요청의 논리블록번호를 다룰 수 있는 블록 계층에서 프리페치 관리자를 동작시킨다. DiskSeen에서 접근되는 모든 블록은 기본적으로 최근의 입출력 번호를 4개까지 저장한다. 입출력 번호는 입출력 발생시 1씩 증가되는 카운터이다. 디스크 요청이 발생하면 주변의 블록들과의 상관관계를 입출력 번호의 차이를 통하여 분석한다. 만약 상관관계가 감지되면 관련 블록들을 논리블록번호 순으로 프리페칭하여 입출력 성능을 높인다. 또, 최대 4개의 입출력 번호를 저장하고 있기 때문에 여러 개의 상관관계 흐름을 감지할 수 있는데, 이 경우에는 판단실패로 인한 오버헤드를 줄이기 위하여 공통적으로 접근되는 블록들만 프리페칭한다. DiskSeen은 접근되지 않는 블록들에 대한 정보를 관리하지 않기 위해서 각 블록의 입출력 번호를 저장하기 위해서 2단계 사상을 이용한다. 이는 1GB 블록의 입출력 번호 관리를 위해서 약 4MB의 메모리 공간이 필요함을 의미한다. 또, 기동시퀀스 전체에 대한 논리블록번호 정렬이 아닌 여러 개의 부분 정렬 패턴의 프리페칭이 수행됨으로 인하여 전체 프리페칭 속도는 낮아진다.

Chang [2] 등은 추측 실행 (speculative execution)을 이용한 프리페

칭 기법을 제안하였다. 추측 실행은 앞으로 실행될 코드를 먼저 실행함으로써 사용될 블록 요청이 미리 발생되도록 유도하는 기법이다. 추측 실행을 위한 프로세스를 생성하고 운영체제에서 프리페쳐 및 대상 프로세스의 블록 요청 패턴을 분석함으로써 프리페쳐 프로세스의 프리페칭 속도를 조절한다. 또, 프리페쳐와 대상 프로세스를 프로세스 수준에서 분리함으로써 대상 프로세스의 논리적인 동작에는 악영향을 주지 않는다. 하지만 현재 확보되지 않은 데이터에 의해 분기해야 하는 경우에는 프리페쳐 프로세스가 정상적으로 수행되지 않으며, 추측 실행은 프로세스가 발생시키는 순서대로 입출력을 요청하기 때문에 입출력 최적화가 어렵다. 이는 하드디스크와 같은 장치에서 효율성을 떨어짐을 의미한다. 휴대장치에서는 프리페쳐 프로세스로 인한 에너지 소모가 높을 수 있기 때문에 추측 실행 기법은 적용에 한계가 있다.

컴파일러 수준에서 정적인 분석을 통하여 프리페치 코드를 삽입함으로써 가까운 시간 내에 요청될 플록을 프리페칭하는 기법이 연구되었다[18][35]. Mowry [18] 등은 처리할 데이터의 크기가 메인 메모리 용량보다 큰 경우, 프리페치 코드를 삽입함으로써 페이지 폴트로 인한 디스크 처리 시간을 단축시키는 기법을 제안하였다. 또, 가까운 시간 내에 사용되지 않는 데이터의 정보를 운영체제에 알려 줌으로써 페이지 교체가 효율적으로 수행되도록 하였다. 이 기법은 프리페치 처리로 인한 오버헤드가 커서, 처리해야 할 데이터의 크기가 장착된 메인 메모리 크기보다 작은 워크로드의 경우, 성능이 향상되는 경우보다 저하되는 경향이 있다. 특히, 논문의 실험에서 접근되는 블록들이 모두 디스크에 캐싱되어 있는 상태에서의 프리페칭은 심각한 성능 저하를 보여줬다. 실험에 사용된 워크로드는 매트릭스 연산 6개, 외부 정렬 1개, 몬테칼로 시뮬레이션 1개로 컴파일러가 쉽게 페이지폴트를 예상할 수 있는 것들만 사용되었다. 이 기법이 응용프로그램 기

동에 적용되기 어려운 점은 어플리케이션 기동 시에 접근되는 블록들의 크기는 메인 메모리의 크기보다 일반적으로 훨씬 작으며, 컴파일러 수준에서 프로그램 기동에 사용되는 블록들을 분석해 내는 것이 어렵고 많은 계산을 요구한다. 또, 이 기법은 응용프로그램의 소스코드가 반드시 필요하다.

VanDeBogart [22] 등은 처리해야할 데이터의 크기가 크고 하드디스크로부터 데이터가 접근되는 패턴을 프로그램의 소스코드를 통하여 쉽게 알 수 있을 경우, 개발자가 앞으로 접근될 영역을 라이브러리 및 운영체제를 통하여 프리페칭하도록 힌트를 주는 코드를 구현하고 이를 콜백 (callback) 방식으로 수행함으로써 디스크 접근 시간을 단축시키는 기법을 제안하였다. 개발자는 디스크 블록의 접근 패턴을 분석하여 앞으로 접근될 블록들을 디스크캐시에 읽어놓도록 요청하는 콜백 함수를 작성하고, 운영체제는 미리 읽어 놓은 블록들을 응용프로그램이 모두 읽어 가면 콜백 함수를 호출하여 다음 접근될 대상을 프리페칭한다. 또, 콜백 함수는 허용된 버퍼 크기가 찰 때까지 프리페치 요청을 할 수 있고, 모여진 요청은 하드디스크의 디스크 헤드 움직임을 최적화시키기 위해서 논리블록번호로 정렬되어 처리된다. 이 기법이 응용프로그램의 기동시간 단축에 사용되기 어려운 점은, 기동시퀀스는 복잡한 패턴을 보이기 때문에, 기동 중에 다음 접근될 블록들을 짧은 시간 내에 계산해 내는 콜백 함수의 작성이 굉장히 어렵다는데 있다.

2.3.3 그 외의 기법들

가전제품의 빠른 기동을 위하여 제조사의 개발자들은 부팅 과정을 제품에 맞게 최적화한다. 예를 들면, 시스템 부팅에 결정적인 영향을 주지 않는 코드를 부팅 이후에 수행되도록 연기하고, 주변 장치가 초기화

되는 동안 유휴상태에 있는 프로세서를 이용하여 부팅에 필요한 처리를 수행하여 시스템의 응답속도를 개선한다. 이러한 노력들은 디지털 TV 또는 디지털 카메라와 같은 가전기기에 많이 적용되고 있다 [10][20]. 이러한 기법들은 개발자가 해당 시스템에 전문적인 지식이 있어야 하고 많은 시간이 소비된다.

하드디스크에서 블록의 배치를 개선함으로써 기동 속도를 향상시키는데 적용할 수 있는 연구들이 있다. 프로그램 기동 시의 하드디스크 헤드 움직임을 줄이기 위해서 기동 시 접근되는 파일들을 연속된 빈 공간으로 이동시키거나 인접한 실린더 그룹들로 구성된 작은 파티션에 옮기는 기법 [1] 등이 제안되었다. 그 외에도 FS2 [42] 와 같이 파일시스템의 빈 공간에 상관관계가 높은 블록들을 접근 순서대로 복제함으로써 하드디스크의 헤드 움직임을 줄여 응용 프로그램의 기동 속도를 개선할 수 있다.

최근, 플래시 기반의 장치를 하드디스크의 캐시로 사용함으로써 플래시 메모리의 임의 접근 성능과 하드디스크의 순차 성능을 하드디스크에 가까운 비용으로 구성하기 위한 기법들이 많이 소개되었다. 하이브리드 하드디스크와 같이 하드디스크 내부에 플래시 메모리를 장착하고 임의 접근 패턴을 보이는 블록들을 플래시에 저장하는 제품들이 출시되었다. 또, 인텔의 터보메모리 [17] 와 같이 작은 용량의 플래시를 하드디스크의 캐시로 이용하거나 Smart Response Technology (SRT) 와 같이 비교적 큰 용량의 SSD를 하드디스크의 캐시로 이용하는 기법들이 실용화되었다. 이와 같은 하이브리드 하드디스크 기법은 플래시에 캐싱될 블록을 선정하고 캐시로부터 방출하는 알고리즘의 성능에 따라 저장장치의 전체 성능이 결정된다.

제 3 장　응용프로그램 기동 시의 동작 특성 분석

응용프로그램이 기동될 때 프로세서 및 디스크의 사용 시간은 두 하드웨어의 성능과 응용프로그램의 자원 사용 패턴에 큰 영향을 받는다. 프리페칭 기법은 프로세서의 시간 보다 디스크 시간을 단축시키는 것에 그 목적이 있기 때문에, 본 장에서는 응용프로그램의 기동 시에 발생되는 디스크 입출력 요청에 대해서 블록 및 파일 수준에서 분석한다.

3.1 기동 시나리오

응용프로그램의 기동시간은 기동 시 읽히는 블록들이 디스크캐시에 적재된 정도에 따라 크게 영향을 받는다. 기동에 필요한 디스크 블록이 디스크캐시에 적재되어 있으면 디스크 입출력을 위해 요구되는 프로세서 및 디스크 사용 시간이 단축된다. 다음은 기동 시 접근되는 모든 블록들이 디스크캐시에 적재되지 않은 시나리오와 모든 블록이 디스크캐시에 적재된 시나리오에 대하여 알아본다.

콜드스타트 (cold start): 콜드스타트는 기동 시에 접근되는 모든 블록이 디스크캐시에 적재되지 않은 상황에서의 응용프로그램 기동을 의미한다. 운영체제 부팅 후 응용프로그램을 처음 기동하는 경우 발생한다. 또, 이 전에 기동을 한 적이 있지만 기동에 사용된 블록들이 다른 응용프로그램에 의해서 디스크캐시로부터 방출된 경우에도 콜드스타트가 발생한다. 큰 파일을 읽거나 복사하는 경우, 그리고 인터넷으로부터 큰 파일을 다운받거나 여러 프로그램을 기동한 경우, 디스크캐시로부터 해당 프로그램의 기동블록들이 디스크캐시로부터 모두 방출되고 콜드스타트

시나리오가 발생한다. 본 논문의 기법은 콜드스타트 상황에서 기동시간을 단축시키는 것을 목표로 한다.

웜스**타트 (warm start)**: 웜스타트는 응용프로그램 기동에 필요한 모든 블록들이 디스크캐시에 적재된 상태에서 응용프로그램이 기동되는 상황을 의미한다. 응용프로그램을 종료한 후 같은 프로그램을 곧바로 실행한 경우에 주로 발생한다. 기동에 필요한 모든 블록들이 디스크캐시에 있으므로 기동 중 디스크 읽기 요청은 발생하지 않는다. 프리페쳐를 사용한 콜드스타트 시간은 웜스타트 시간보다 짧을 수 없다.

3.2 기동 시 발생되는 디스크 입출력 분석

디스크의 처리 성능은 입출력 패턴에 큰 영향을 받기 때문에 기동 블록 과 디스크의 특성 분석은 디스크 프리페치 최적화 기법을 연구하는데 중요한 정보가 된다. 본 절에서는 응용프로그램 기동 시에 요청되는 디스크 입출력 요청 개수, 요청 크기 및 읽기 요청의 발생 비율을 알아본다.

프리페칭은 가까운 시간 내에 응용프로그램으로부터 읽기 요청이 예상되는 블록을 디스크캐시에 미리 읽어 놓음으로써 프로세스의 디스크 요청에 대한 응답시간을 단축시키는 효과가 있다. 따라서 응용프로그램이 기동될 때 읽기 요청이 많을수록 프리페쳐의 효율성이 높아질 수 있다. 표 1은 본 논문에서 성능평가를 위해 사용한 14개의 데스크탑 PC용 응용프로그램이 각각 기동될 때 발생하는 입출력 요청의 종류, 개수, 크기에 대한 정보를 나타낸다. 리눅스 응용프로그램이 기동될 때 읽기 요청의 비율이 압도적으로 많음을 알 수 있다.

표 1 응용프로그램 기동 시 읽기 및 쓰기 I/O 요청의 개수와 총 크기

응용프로그램	읽기 I/O 요청의 수	읽기 I/O 요청의 크기 (KB)	쓰기 I/O 요청의 수	쓰기 I/O 요청의 크기 (KB)
Acrobat Reader	906	51,208	4	16
Designer-qt 4	2,157	54,472	23	92
Eclipse	4,764	97,460	32	128
Firefox	834	42,732	0	0
Gimp	1,807	51,480	87	348
Houdini	4,117	165,448	244	976
Kdevelop	3,601	126,012	95	380
Konqueror	1,811	41,532	2	8
Labview	2,123	79,416	253	1,012
Libreoffice	940	55,824	63	268
Matlab	4,457	153,008	316	1,264
Scribus	2,967	98,488	21	96
Thunderbird	749	65,048	33	132
XilinxISE	2,869	163,036	151	612

표 1의 데이터는 Fedora 16 64비트 버전에 기본적으로 설치되어 있는 응용프로그램과 추가적으로 설치한 응용프로그램으로부터 수집하였다. 파티션은 ext4로 관리되며 아이노드 미리읽기는 수행하지 않도록 설정하였다. 테스트에 사용된 응용프로그램에서 읽기요청의 입출력 수는 Thunderbird의 기동 시 749개로 가장 적었고, Eclipse 기동 시 4764개로 가장 많았다. 읽기 요청의 총 크기는 Konqueror가 약 42MB로 가장 작았고, Houdini가 약 165MB로 가장 컸다. 기동 블록의 특성은 운영체제, 파일시스템, 그리고 응용프로그램에 따라 차이가 있다 [41].

3.3 프로세서와 디스크의 활성화 패턴 분석

하나의 물리적인 칩에 여러 개의 논리적인 프로세서 코어를 탑재하는 멀티코어 기술은 서버 뿐 아니라 모바일 폰 용 프로세서에도 적용된 지 오래되었고, 프로세서의 클럭 속도 향상 폭이 둔화되면서 프로세서의 성능 향상을 위한 주된 기법이 되어 왔다. 또, 플래시 기반의 저장장치는 내부적으로 여러 개의 플래시 칩을 내장하고 이를 병렬적으로 활용함으로써 처리율을 높이고 있다.

그림 4는 인텔 I7-860 프로세서와 인텔 G2 80GB를 사용하는 환경에서 Eclipse의 콜드스타트 시 프로세서와 SSD의 사용 여부를 1ms 주기로 샘플링한 결과를 보여준다. 전체 4738개의 샘플 중 109개의 샘플

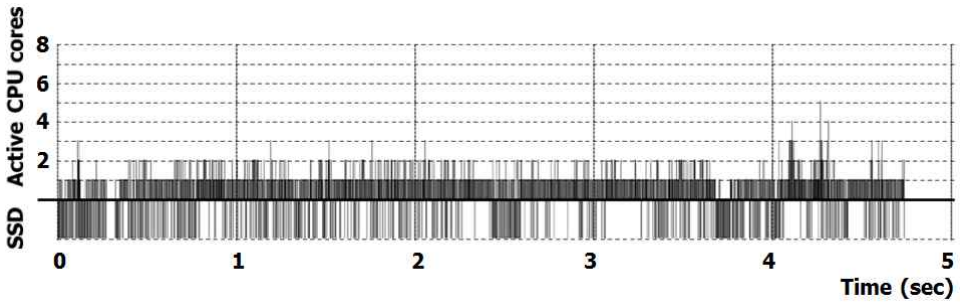

그림 4 Eclipse 기동 시 프로세서와 SSD의 사용 패턴

(sampling rate: 1 KHz)

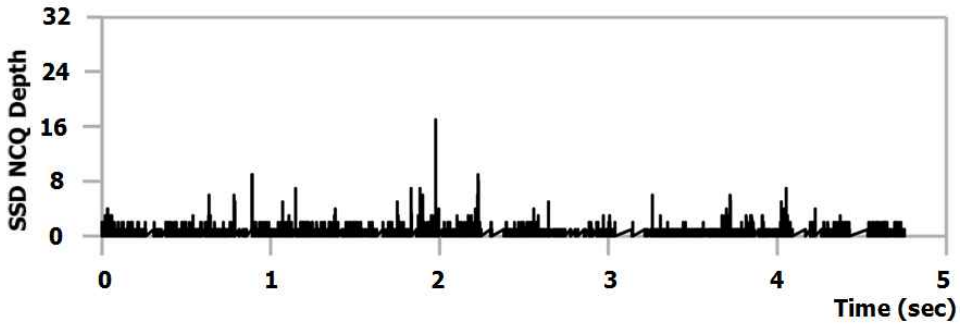

그림 5 Eclipse 기동 시 SSD에서 동시에 처리 중인 요청의 개수

에서만 프로세서와 SSD가 동시에 활성화 되었다. 또, 적어도 1개의 프로세서 코어가 활성화된 3685 샘플 중 1개의 코어만이 사용된 샘플의 수가 3623개로 멀티코어 활용률이 낮음을 알 수 있었다. 성능평가에 사용된 다른 프로그램들도 Eclipse와 마찬가지로 멀티코어의 활용이나, 프로세서와 SSD의 병렬화는 거의 이루어지지 않았다. 이는 프로그램 기동 작업을 여러 개의 세부 작업으로 나누어 병렬적으로 수행하는 것이 어려운 경우가 많고, 프로세스의 동기적인 디스크 입출력으로 인하여 SSD와 프로세서가 동시에 사용되지 못하기 때문이다. 그림 5는 Eclipse의 콜드 스타트 시 SSD의 명령큐에 저장된 명령어의 수를 나타낸다. Eclipse 기동 시 SSD에서 동시에 처리되고 있는 명령어의 수는 평균 0.3 정도로 낮았다.

그림 6은 테스트에 사용된 응용프로그램이 기동될 때 발생하는 블록 요청을 크기에 따른 비율로 도시한 그래프이다. 그림에서 보여주듯이 4KB 이하의 작은 요청이 기동시퀀스의 큰 비중을 차지한다.

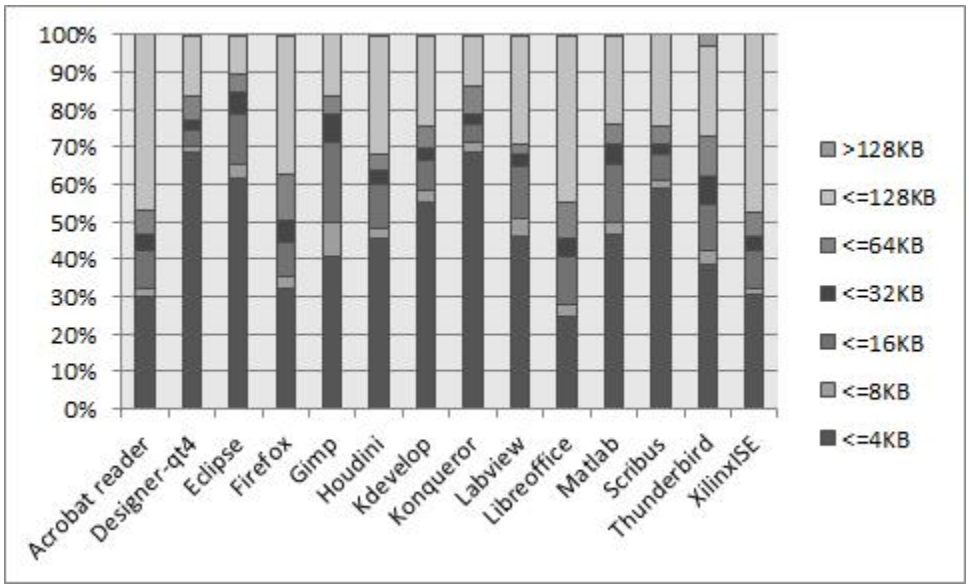

그림 6 응용프로그램의 기동 시 발생하는 블록 요청의 크기별 비율

Libreoffice 기동 시 4KB 요청의 비율이 24.5%로 가장 작고 Designer-qt4 기동 시 69%로 가장 컸다. 또 128KB를 초과하는 큰 요청은 거의 발생하지 않았다. 응용 프로그램의 기동과 같이 많은 수의 작은 입출력 요청들이 동기적으로 발생하는 하는 경우, SSD 내부병렬성의 제한적인 이용 때문에 SSD의 처리율이 낮다.

하드디스크의 경우, 메인라인 제품의 분당회전속도 (revolution per minute, RPM)는 5400~7200 RPM으로 과거에 비해 향상이 거의 없다 [11]. 서버용 제품군에는 10000~15000 RPM의 빠른 회전 속도를 갖는 제품이 있지만 가격이 높고, SSD에 비해 성능이 낮아 제품 경쟁력이 떨어진다. 또, 기동 시 디스크 요청은 임의 접근 패턴에 가깝기 때문에 큰 디스크 헤드의 움직임이 많이 발생한다. 이는 탐색 시간의 단축을 통하여 개선될 수 있지만, 탐색 시간은 굉장히 느리게 개선되어 왔다 [11]. 반면, 하드디스크는 집적 능력 향상을 통하여 순차 처리 속도는 점점 개선되고 있고, NCQ 지원 하드에서는 명령큐 내의 요청 순서 변경을 통하여 전체 헤드의 움직임은 단축될 수 있다. 하지만, 콜드스타트 시에 명령큐 내의 요청 개수는 낮기 때문에 NCQ 기능을 잘 활용하고 못한다. 프리페쳐를 통하여 비동기적으로 많은 수의 명령을 동시에 요청함으로써 NCQ에서 제공하는 장점들을 이용하는 것이 중요하다.

제 4 장 커널 수준 실행시간 프리페쳐의 설계, 구현 및 평가

4.1 실행시간 프리페쳐의 소개 및 목표

응용프로그램은 실행파일 로더를 통하여 수행되기 때문에 기동 시점을 쉽게 알 수 있다. 이 때문에 범용 워크로드 프리페칭 기법들에서 중요한 과제로 여겨지는 블록 간 상관관계 분석 대한 오버헤드를 크게 낮출 수 있다. 실행파일 로더는 기동 블록들의 수집, 프리페치 수행 등의 동작 단계를 파악하고 관련 처리를 하는데 적절한 지점이다. 기동 블록의 정확한 수집과, 수집된 기동시퀀스를 빠르게 프리페칭하기 위해 요청 순서를 최적화하는 작업은 프리페쳐의 성능에 큰 영향을 준다.

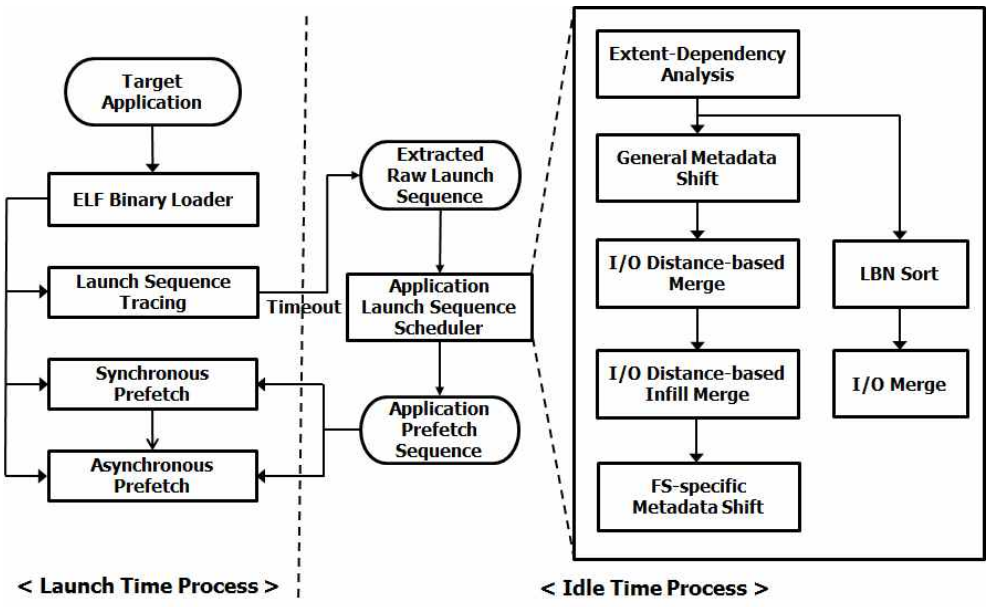

그림 7 제안한 실행시간 프리페쳐의 프레임워크

그림 7은 본 논문에서 제안한 커널 수준 프리페쳐 (Paralfetch)의 구조를 나타낸다. Paralfetch에서는 기동 단계를 기록하기 위하여 실행파일의 빈 공간을 사용하였다. 리눅스의 ELF 바이너리에서 ELF 헤더의 첫 16 바이트 중 8 바이트는 패딩 (padding)을 위하여 사용된다 [71]. Paralfetch에서는 패딩 바이트 중 첫 1 바이트를 사용하여 프리페치 단계를 표시하였다. 실행파일에 프리페치 단계를 표시함으로써 응용프로그램 이름과 프리페치 단계를 조사하는 과정이 필요 없고, 기동 단계 저장을 위한 별도의 저장공간을 요구하지 않는다. 윈도우즈 프리페쳐의 경우 프리페치 단계 관리의 오버헤드를 줄이기 위해서 프리페치 대상 응용프로그램의 수를 128개로 제한하고 있고, 구글-프리페치에서는 프로그램 실행 시 해시와 리스트를 이용하여 응용프로그램의 프리페치 단계를 검사한다. 프리페치의 단계는 프리페치시퀀스 생성, 프리페치 유효성 검사, 프리페치 활성 및 비활성, 프리페치 순서 최적화 단계가 있다. 상태의 전이는 자동 모드와 수동 모드에 따라 다르며 그림 8과 같다.

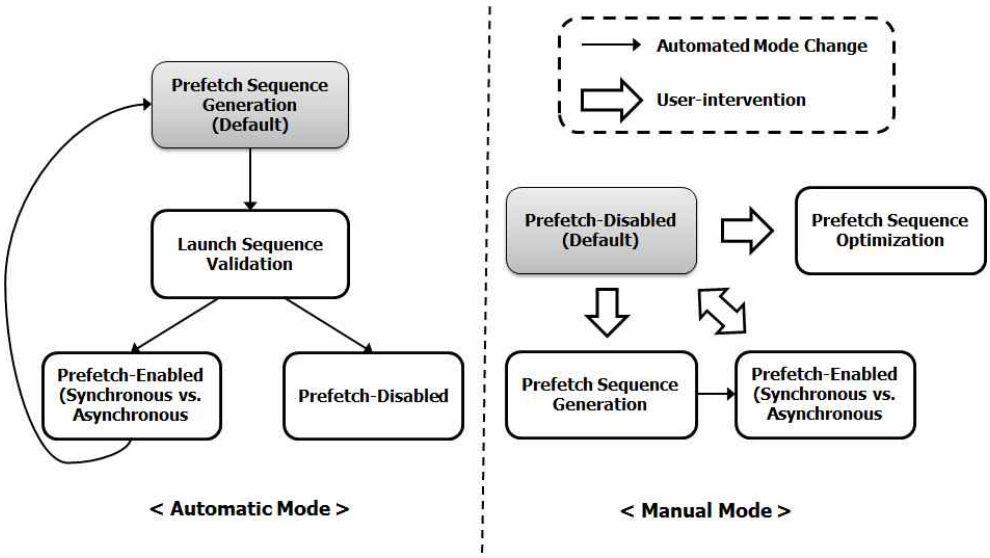

그림 8 프리페치 단계 전이도

ELF 바이너리의 패딩 영역은 기본적으로 0으로 채워져 있다. 수동 모드에서 0은 프리페치 비활성화를 의미하고 자동 모드에서 0은 프리페치 정보 파일 생성 단계를 의미한다. 자동 모드에서는 프리페치 정보 파일 생성 이후 한 번 더 프리페치 정보를 생성하여 두 프리페치 시퀀스 정보의 유사도를 비교한다. 이 때 공통적으로 수집된 요청의 개수나 총 입출력 크기가 임계값 보다 작은 경우 프리페치 비활성화 단계로 전이된다.

시스템 파티션으로 하드디스크와 같이 느린 장치가 사용되는 경우에는 자동 모드가 기본 모드로 설정된다. 하지만 시스템 파티션에 SSD가 사용되는 경우에는 수동 모드가 기본 모드로 설정된다. SSD를 사용하면 기동시간을 하드디스크에 비하여 크게 단축시킬 수 있고 운영체제에 따라서 프리페치로 인한 기동시간 단축이 작을 수 있기 때문에, 이 경우에는 기동시간이 비교적 긴 응용프로그램만 프리페치를 적용시키기 위하여 수동 모드를 기본 모드로 한다. 수동 모드에서는 사용자가 Parafetch를 쉽게 적용 할 수 있도록 nautilus-actions 패키지를 이용하여 X-Windows와 연동시켰다. nautilus-actions는 응용프로그램 아이콘의 마우스 오른쪽 버튼 클릭 메뉴를 통하여 특정 작업을 수행 할 수 있도록 인터페이스를 설정하는 툴이다.

마우스 오른쪽 버튼 클릭 메뉴에 “Prefetch Enable”과 “Prefetch Disable” 두 가지 메뉴가 나타나고 각 항목을 선택하였을 때 아이콘 클릭 시 수행되는 바이너리 파일의 패딩 첫 바이트의 값이 프리페치를 활성화 또는 비활성화하는 값으로 변경된다. 프리페치 활성화 선택 시에 프리페치 정보 파일 (프리페치시퀀스 파일)이 없는 경우에는 프리페치시퀀스 생성 단계로 설정된다.

프리페치시퀀스 생성 단계에서는 응용프로그램 기동 중 디스크캐시 미스 (miss)로 인하여 생성되는 디스크 읽기 요청 정보가 로그에 저장된다.

프로그램 실행 후 정해진 시간이 지나거나 읽기 요청의 빈도가 정해진 임계치보다 낮으면 기동이 끝났다고 판단하고, 해당 부분까지의 로그는 기동시퀀스 스케쥴러를 통하여 최적화가 수행된다. 이 후 최적화된 프리페치시퀀스 정보는 파일로 저장되어 이 후에 응용프로그램 기동 시 프리페치 수행을 위해 사용된다. 프리페치는 동기 및 비동기로 수행되는데 디스크의 특성에 따라 결정된다.

표 2 프리페치 단계

프리페치 단계	설명
기동시퀀스 생성 (Launch Sequence Generation)	응용 프로그램의 기동시퀀스 생성 단계로 기동 시 접근된 블록들의 정보를 수집하여 이를 최적화하고 프리페치시퀀스 정보 파일로 저장하는 단계
기동시퀀스 유효화 검사 (Launch Sequence Validation)	기동시퀀스의 유효성 검사를 위해 기동시퀀스를 수집하여 기존의 프리페치시퀀스와의 유사도 비교를 하는 단계. 유사도가 임계치 보다 낮은 경우 해당 응용프로그램의 프리페치가 비활성화 될 수 있음
프리페치 활성화 (Prefetch-Enabled)	프리페치시퀀스 파일이 생성되어 기동 시 프리페치가 수행되는 단계. 디스크의 특성에 따라 동기, 비동기 또는 두 가지가 섞인 방식으로 프리페치가 수행됨
프리페치 비활성화 (Prefetch-Disabled)	프리페치를 수행하지 않는 단계
프리페치시퀀스 최적화 (Prefetch Sequence Optimization)	동기적, 비동기적 프리페치가 조합된 경우에 동기적으로 프리페치되는 블록을 가능하면 비동기적으로 프리페치 되도록 이동시켜 프로세서와 디스크의 병렬적 수행 구간을 늘림. 회전하는 매체를 포함한 디스크에 적용 가능

Paralfetch에서는 실용성 (practicality)를 높이기 위해서 프리페치를 통한 성능 향상 뿐 아니라 오버헤드 줄이기 위하여 많은 노력을 하였다 [64][65]. 다음으로 프리페쳐의 각 부분에 대한 설계 및 구현에 대해서 알아보고 기존의 기법들과도 비교하도록 하겠다.

4.2 기동시퀀스 수집

프리페칭 기법은 가까운 미래에 요청될 블록을 정확히 예측하는 것이 중요하다. 구글-프리페치나 윈도우즈 프리페쳐에서는 정규파일의 접근 정보를 수집한다. Paralfetch에서는 기동 블록의 정확한 수집을 위하여 정규파일 뿐 아니라 inode 블록, indirect 블록, 디렉토리 엔트리 (dentry) 블록 등과 같은 메타데이터에 대한 요청도 수집 및 프리페치 대상에 포함하였다. 또, 콜드스타트 시나리오를 재현하기 위하여 응용프로그램 기동 전 디스크 캐시를 무효화시킨 후에 기동 관련 블록 정보를 수집한다.

버퍼캐시와 페이지캐시로 구성된 디스크캐시 계층에서 읽기 요청 블록의 탐색에 실패한 후, 블록 레이어로 디스크 입출력 처리를 요청할 때 로그를 남기도록 구현하였다. Inode나 디렉토리 엔트리 블록의 경우 slab 할당자를 통하여 오브젝트 단위로 중복 캐싱된다. 본 논문에서는 아이노드 및 디렉토리 엔트리와 같이 디스크 블록의 캐싱을 위해 사용되는 슬랩 할당자를 slab 캐시라는 용어로 표현하도록 하겠다. slab 할당자의 무효화는 불완전하기 때문에 본 논문에서 사용한 기동시퀀스 수집 방식은 완벽한 기동시퀀스를 수집하지는 못한다.

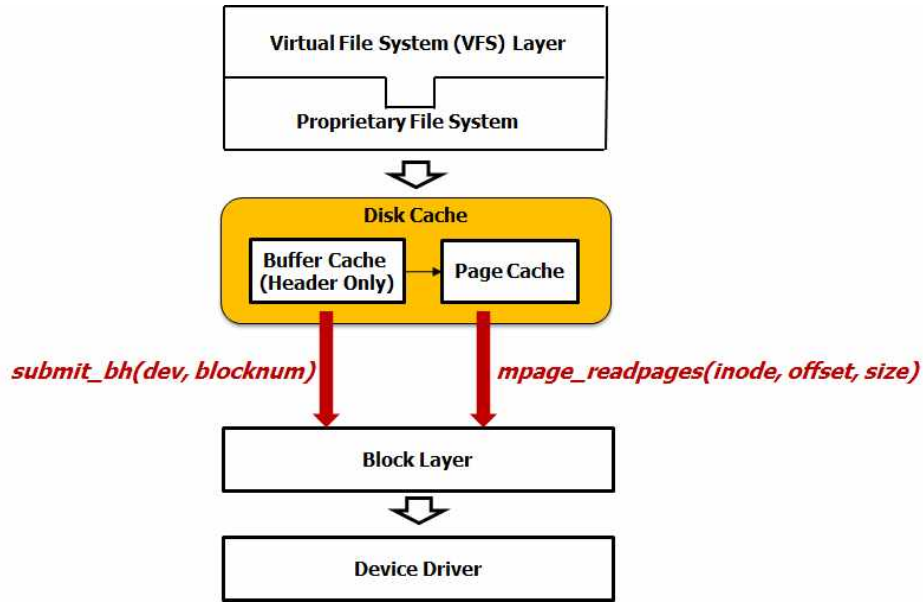

그림 9 기동시퀀스 로깅이 수행되는 함수와 디스크 입출력 계층

그림 9는 디스크 입출력 계층과, 디스크캐시에서 탐색에 실패하여 블록 레이어로 입출력 요청을 수행하는 함수를 나타낸다. 제안한 프리페처에서 버퍼캐시 탐색 실패로 인해 디스크 입출력을 요청할 때 호출하는 submit_bh 함수와 페이지캐시에서 탐색 실패로 인해 호출되는 mpage_readpage (또는 mpage_readpages)에서 요청 블록의 로그를 저장하도록 하였다. 저장되는 정보의 구조체와 그 멤버 변수는 표 3과 같다.

표 3 기동시퀀스 로그 정보

자료형	변수명	설명
u64	ts	입출력 발생 시간 (나노초 단위)
dev_t	dev	장치 번호
ino_t	ino	페이지캐시 처리의 경우 해당 아이노드 번호, 버퍼캐시의 경우 0
u64	blk_num	블록 시작 번호
u32	blk_len	요청 블록의 크기

기동 블록의 정보는 $T_{timeout}$에 해당하는 시간만큼 수집되며, 이후에는 더 이상 로그가 기록되지 않도록 로그 처리를 비활성화한다. 응용프로그램의 실제 기동시간은 $T_{timeout}$보다 길거나 짧을 수 있다. 실제 기동시간이 $T_{timeout}$보다 짧은 경우 기동 중에 발생한 요청들을 구분하기 위하여 요청 간 시간 (inter-arrival time)과 초당 요청 개수를 분석하여 기동의 마지막 블록을 예측한다. 따라서 기동 블록이 충분히 수집될 수 있는 $T_{timeout}$ 값을 사용하여야 한다. 테스트 프로그램 중 가장 긴 요청 간 시간은 697ms였기 때문에 제안한 프리페처에서는 1초로 설정하였다. 또, 초당 요청 개수가 5개 미만일 경우 기동 완료로 판단하도록 설정하였다. 테스트 프로그램에서 초당 요청 개수가 최소인 경우는 8개였다. $T_{timeout}$의 경우 디스크의 성능과 프로세스의 성능에 크게 좌우된다. 하지만 본 논문에서는 타임아웃 값 뿐 아니라 읽기 입출력 요청의 패턴을 분석을 통하여 기동 기간을 예측하기 때문에 $T_{timeout}$은 충분히 큰 값으로 초기화된다. 하드디스크가 장착된 시스템의 경우 30초, SSD가 장착된 시스템의 경우 10초, 모바일 플랫폼의 경우 20초로 $T_{timeout}$값을 초기화하였고 이 값은 proc 파일을 통하여 변경 가능하다.

4.3 프리페치시퀀스 스케쥴러

4.3.1 익스텐트-의존성 (Extent-Dependency) 분석

리눅스 커널이 제공하는 디스크캐시 무효화 처리는 완벽한 콜드스타트를 재현시켜 주지 못한다. 이유는 디스크캐시는 거의 완벽하게 무효화가 되지만, inode 및 디렉토리 엔트리 오브젝트의 slab은 완벽하게 무효화가 되었을 경우에 시스템의 전체적인 성능저하를 유발할 수 있기 때문에 완벽하게 무효화되지는 않는다. Paralfetch에서는 slab을 통해 할당된 오브젝트에서 히트가 발생함으로 인해 디스크 엑세스가 발생하지 않은 블록들 (디스크캐시에는 없지만 slab 캐시에 존재하는 블록)을 파악하기 위하여 파일시스템 수준의 의존성 검사를 수행한다.

익스텐트 (extent)는 파일 및 논리블록번호 상에서 연속적인 가장 큰 블록 집합을 의미한다. 일반적으로 파일시스템은 블록 단위 또는 익스텐트 단위로 블록 의존성을 갖을 수 있다. 익스텐트-의존성 분석기에서 의미하는 익스텐트는 기존의 익스텐트 분해 조건 만족하는 동시에 같은 블

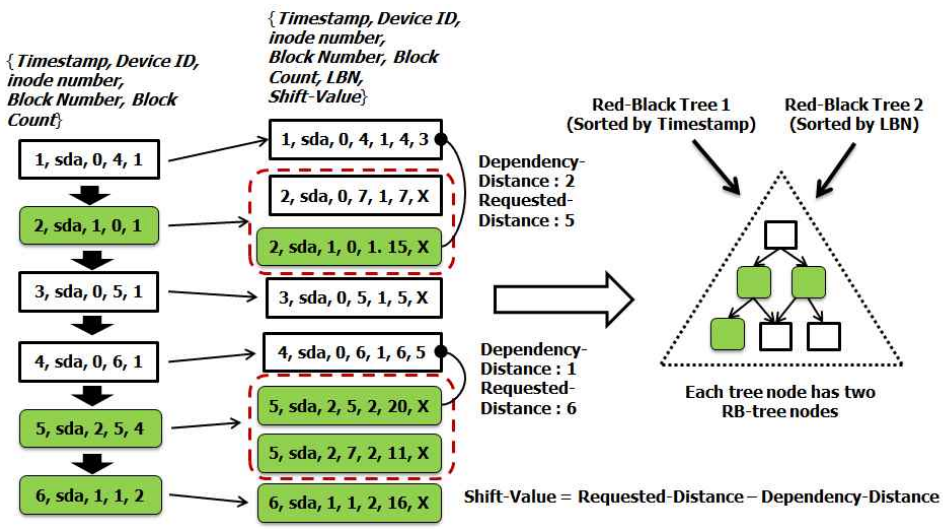

그림 10 익스텐트-의존성 분석

록에 의존성을 갖는 가장 큰 블록 집단을 의미한다. 예를 들어, 파일 및 논리블록 번호 상으로는 연속적이어도 서로 다른 블록에 의존성을 갖는다면 이들은 서로 다른 익스텐트로 분리한다.

그림 10은 익스텐트-의존성 분석의 과정을 나타낸다. 첫 단계로 수집된 기동시퀀스 블록들의 의존성 관계를 분석하여 익스텐트를 구분하고, 각각에 대한 익스텐트 노드 정보를 만든다. 또, slab 캐시의 불완전한 무효화로 인하여 수집에 실패한 아이노드 블록이 존재하면 이 정보를 새로운 노드를 만들어 추가한다. 마지막으로 각 노드는 타임스탬프와 논리블록번호로 정렬되는 두 개의 레드-블랙 트리에 연결된다.

대부분의 파일시스템은 블록이나 의존성 개념이 없는 익스텐트에 대한 논리블록번호 사상 정보를 유저영역에 제공하기 위하여 fiemap 또는 fibmap 기능을 제공한다. 이 함수들은 파일 상의 블록 사상 정보를 제공하기 위하여 의존된 블록들을 분석하고 접근한다. 이 때문에, 파일시스템 코드에 구현되어 있는 fiemap 또는 fibmap 함수를 이용하면 익스텐트-의존성 분석 기능을 어렵지 않게 구현할 수 있다.

4.3.2 블록 간 의존성 해결을 위한 메타데이터 쉬프트

플래시 기반의 저장장치는 접근시간이 거의 일정하기 때문에 하드디스크와 같이 헤드의 움직임을 최적화시킬 필요가 없다. 일반적으로 데스크탑이나 노트북에 사용되는 SSD는 8~20개의 플래시 칩으로 구성되고, SATA2 호환 인터페이스를 통한 명령 큐잉을 지원한다. 단일 칩으로 구성된 경우에도 칩 내부의 다이, 플레인 수준에서 병렬적인 처리를 지원한다. 또, 스마트폰이나 스마트패드에 많이 사용되는 eMMC 인터페이스의 차세대 방식인 UFS에서는 공식적으로 명령 큐잉을 지원한다.

SSD의 처리율은 SSD를 구성하고 있는 플래시 칩의 병렬적 사용 정도에 의해 결정된다. 플래시 칩의 내부병렬성을 높이기 위해서는 1) 큰 요청을 통하여 플래시 칩의 병렬적 사용을 유도하거나 2) 명령 큐잉을 이용하여 여러 개의 작은 요청을 동시에 처리함으로써 여러 플래시 칩 그리고 칩 내부의 병렬성을 유도할 수 있다. 응용프로그램의 기동 과정은 많은 수의 작은 디스크 요청을 포함하고 있고 인접하지 않은 경우가 많기 때문에 병합을 통해 요청 크기를 키우는데 한계가 있다. 따라서 명령 큐잉을 활용하여 SSD의 내부병렬성을 높이는 것이 중요하다.

블록 간 의존성 해결을 통하여 SSD의 내부병렬성을 유도하기 위하여, 의존성 관계에 있는 블록 사이에 이 블록들과 의존성이 없는 블록을 채우는 메타데이터 쉬프트 기법을 고안하였다. 메타데이터 쉬프트는 범용 메타데이터 쉬프트 또는 파일시스템 전용 메타데이터 쉬프트를 이용할 수 있다. 범용 메타데이터 쉬프트는 파일시스템에서 익스텐트-의존성 분석을 통하여 Shift-Value가 계산된 상태에서 Shift-Value 만큼 해당 블록을 왼쪽으로 이동시킴으로써 의존성을 원하는 만큼 완화시킨다. 또, 파일시스템이 전용의 메타데이터 쉬프트 함수를 제공하는 경우에는 전용

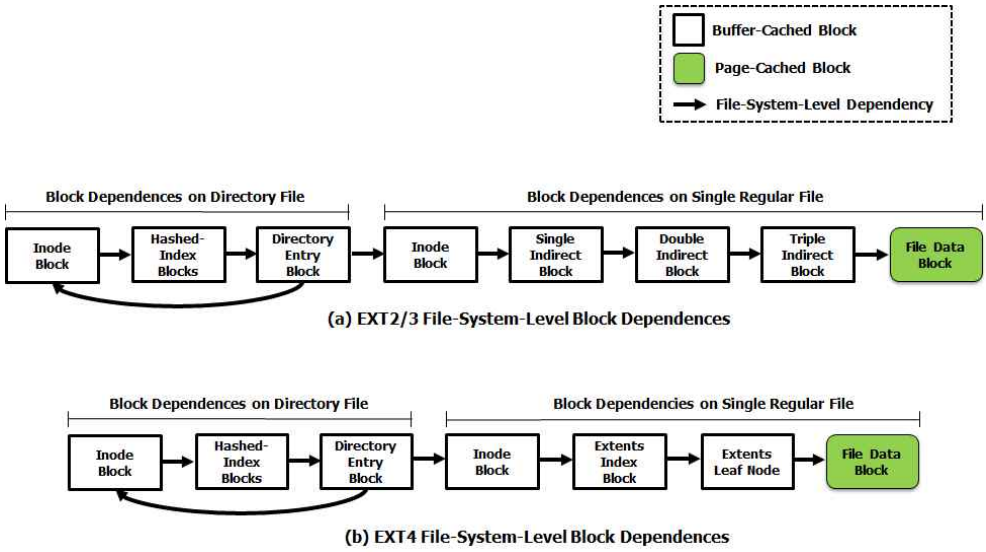

그림 11 파일시스템 수준의 의존성

함수를 이용한다.

그림 11은 리눅스에서 가장 많이 사용되는 ext 계열의 파일시스템에서 파일시스템 수준의 블록 의존성을 나타낸다. 화살표는 의존성을 나타내는데, 보통 의존된 블록이 해당 블록의 위치 정보를 가지고 있기 때문에 발생한다. 화살표 왼쪽 (화살표 시작 부분)의 블록이 준비된 이후에 화살표 오른쪽의 블록 위치를 알 수 있고, 이 때문에 화살표 이후의 블록이 요청될 수 있다.

프리페칭은 디스크 블록을 메인 메모리 상의 디스크캐시에 미리읽는 기법으로서, 이때의 의존성은 파일시스템 수준의 의존성과는 다르고 운영체제에서 지원하는 디스크캐시의 방식에 따라서도 다르다. 그림 12는 리눅스에서 ext 계열 파일시스템의 프리페치 수준 의존성을 나타낸다. ext 계열 파일시스템의 메타데이터는 버퍼캐시로 관리가 되는데, 장치 번호와 장치 내 블록 번호를 알면 버퍼캐시로 캐싱이 될 수 있다. 기동

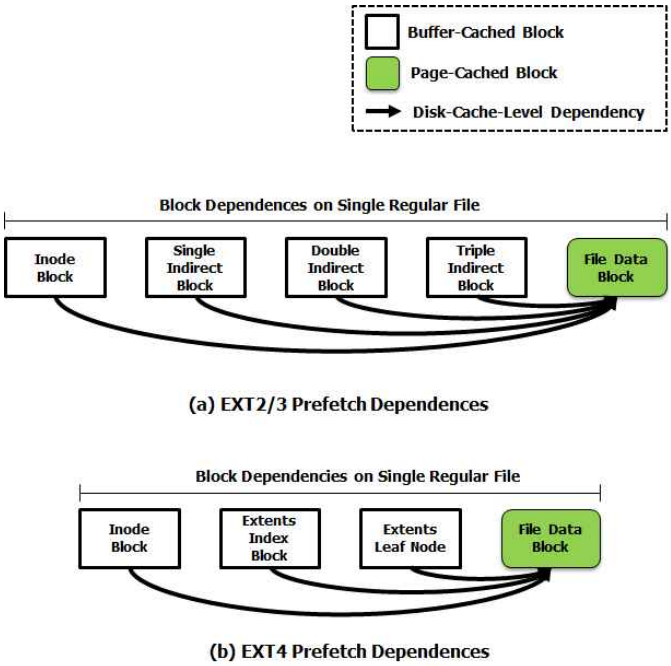

그림 12 프리페치 수준의 의존성

시퀀스 모니터링을 통하여 이 정보들을 이미 알고 있기 때문에 기동시퀀스 프리페칭에서 버퍼캐시로 관리되는 블록들 간에는 의존성이 없다. 하지만 페이지캐시로 관리가 되는 정규 파일의 데이터 블록은 관련된 메타데이터가 모두 준비되어야만 프리페칭이 가능하다. 프리페칭을 비동기적으로 요청함으로써 명령 큐잉을 이용할 수 있지만 프리페치 수준의 의존성으로 인해 큐잉되는 명령의 수가 제한될 수 있다.

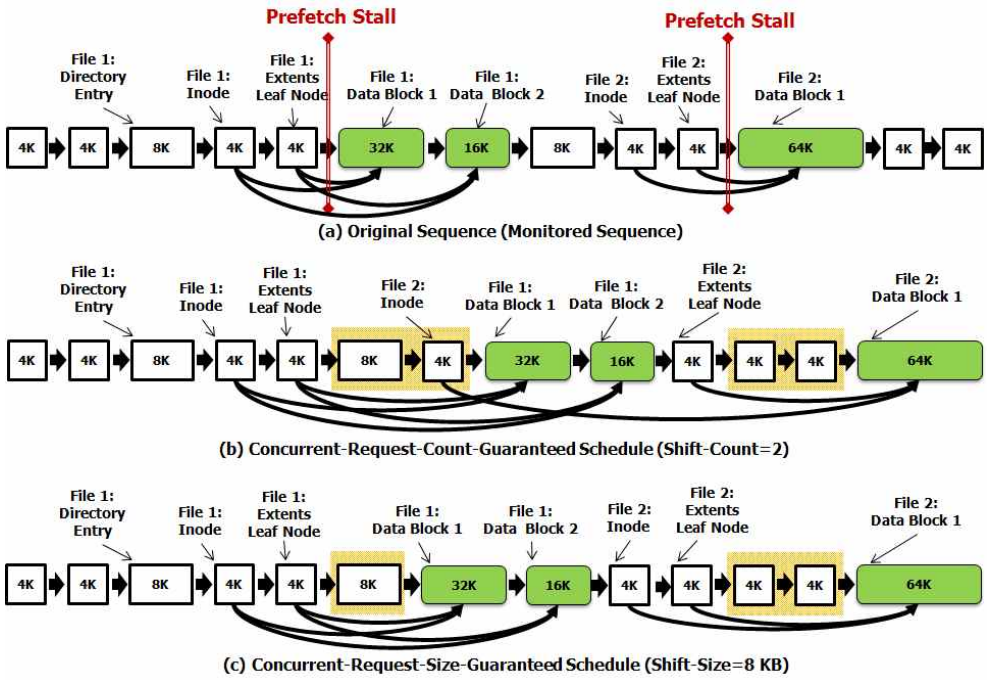

그림 13 메타데이터 쉬프트를 이용한 프리페치 의존성 해결
(ext4 파일시스템의 예)

그림 13-(a)는 수집된 기동시퀀스를 순서대로 프리페칭할 때 발생되는 프리페치 스톨 (prefetch stall)을 보여준다. 기동시퀀스를 비동기적으로 요청하더라도 프리페치 의존성으로 인하여 의존된 블록이 모두 읽혀질 때 까지 추가적인 프리페치가 정지된다. 이를 방지하기 위해서 메타데이터 쉬프트 기법을 고안하였다. 그림 13-(b)는 의존성이 있는 두 요청 사이에 이들과 의존성이 없는 블록 요청을 뒤쪽에서 가져옴으로써, 의존성이 존재하는 두 요청 사이에 존재하는 의존성 없는 요청의 개수를 보장하는 방식이다. 실제로, SSD의 내부병렬성은 요청의 개수보다 명령큐에 들어있는 요청들의 총 요청 크기와 연관성이 더 크기 때문에 그림 13-(c)와 같이 의존성 있는 블록들 사이에 의존성 없는 블록들의 총 요

청 크기를 보장하는 방식을 사용하는 것이 바람직할 수 있다.

4.3.3 거리기반 병합

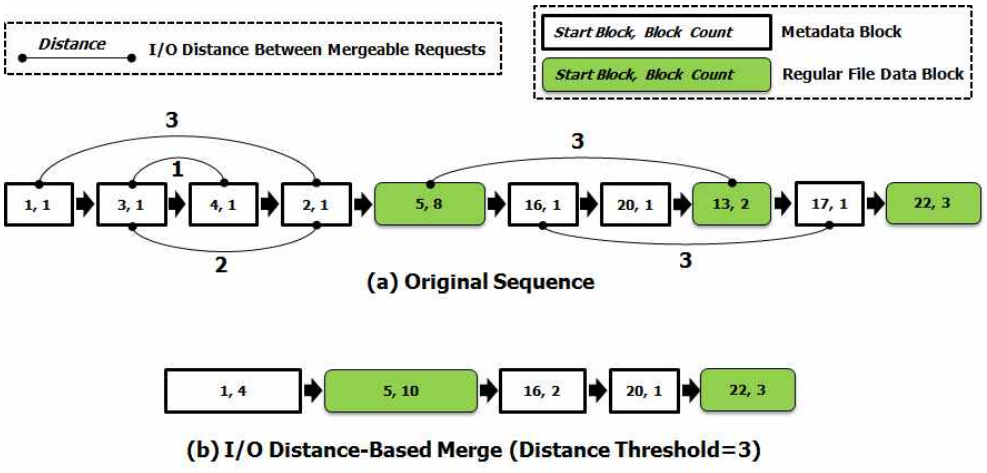

그림 14 거리기반 병합

그림 14는 기동 블록의 거리기반 병합을 보여준다. 기동에 사용되는 블록의 병합이 가능한 경우, 병합을 통하여 하나의 큰 요청으로 처리하는 것이 두 명령을 명령 큐잉을 이용하여 내부병렬화를 유도하는 것보다 일반적으로 높은 처리율을 보여준다. 하지만 플래시 기반의 장치에서는 프리페쳐 쓰레드와 응용프로그램이 동시에 수행되도록 하기 위하여 기동시 요청되는 블록 순서를 유지하는 것이 중요하다. 따라서, 거리기반 병합에서는 병합 대상 블록들이 요청 순서 상 거리가 가까우면서 논리블록번호 상 인접한 경우에 병합을 시도한다. NCQ 미지원 SSD에서는 비동기적으로 입출력을 요청하는 방식으로 SSD의 내부병렬성을 유도할 수 없기 때문에 병합을 효과적으로 이용하여 프리페치 속도를 빠르게 하는 것이 중요하다. 거리기반 병합에서 블록 간 인접성은 논리블록번호로 정렬된 레드-블랙 트리를 통하여 알 수 있고, 두 블록이 인접한 경우 타임

스탬프로 정렬된 레드-블랙 트리에서 두 인접한 요청 간의 입출력 순서 거리를 알 수 있다.

4.3.4 거리기반 빈공간채움 병합

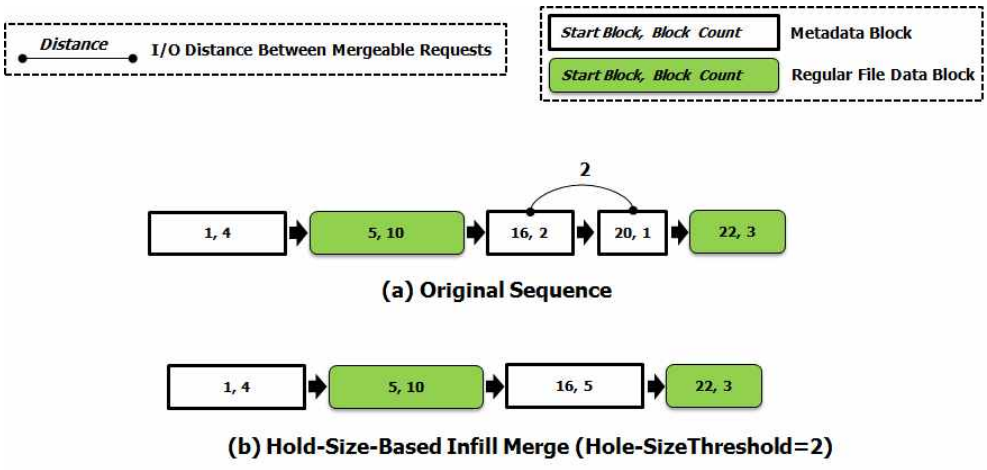

그림 15 거리기반 빈공간 채움 병합

그림 15와 같이 빈공간 채움 병합은 병합 대상 블록 사이에 작은 비요청 공간이 있어도 빈 공간 (hole)을 포함시켜 병합하는 기법이다. 불필요한 영역을 부가적으로 프리페칭하는 대신 NCQ미지원 SSD에서 요청 크기를 늘려 처리율을 높일 수 있다. 또, 빈공간 채움 병합을 통하여 기동블록 수집에 실패한 디렉토리 블록들을 프리페칭하는 효과가 있을 수 있다. 대부분의 파일시스템에서 디렉토리 파일의 데이터블록은 해싱된인덱스 (hashed-index)로 관리가 되기 때문에 디렉토리의 일부를 탐색하는 경우 작은 hole을 포함시켜 병합함으로써 입출력 수를 줄일 수 있다. 거리기반 빈공간 채움 병합에서 논리블록번호상 가장 가까운 노드를 찾아내기 위하여 익스텐트-의존성 검사 과정에서 생성한 논리블록번호 정렬 레드-블랙 트리를 이용하고 두 노드 사이의 기동시퀀스 상 입

출력 순서 거리를 계산하기 위하여 타임스탬프로 정렬된 레드-블랙 트리를 이용한다.

4.3.5 논리블록번호 정렬

하드디스크의 경우, 논리블록번호로 정렬된 순서로 기동시퀀스를 프리페칭하는 것이 디스크 헤드의 움직임을 줄이는데 도움이 된다. 기동시퀀스는 순차 접근 패턴을 보이지 않기 때문에 실제로는 많은 디스크 헤드의 움직임이 발생한다. 기존의 어플리케이션 프리페쳐는 파일 수준의 정렬을 수행하지만 이를 더 최적화하기 위해 본 논문에서는 논리블록번호 정렬 방식을 이용하였다. 익스텐트-의존성 분석 과정에서 각 요청들은 논리블록번호로 정렬된 레드-블랙 트리에 연결되기 때문에 레드-블랙트리를 오름차순으로 순회하면서 프리페치시퀀스를 생성하면 논리블록번호로 정렬된 시퀀스가 생성된다. 이 후, 논리블록번호 상 연속된 같은 종류의 블록 요청들은 병합하여 프리페치 시 입출력 처리율을 높인다.

4.3.6 플러그/언플러그

리눅스에서 파일시스템은 bio 구조체를 이용하여 블록 계층으로 처리를 요청한다 [57][61]. bio는 논리블록번호 상 연속된 디스크 요청을 표현하는 구조체이다. bio 구조체는 디스크 장치 드라이버의 처리 단위인 request 구조체 내의 bio 연결 리스트 멤버 변수에 연결되어 I/O 스케쥴러에 입력된다.

리눅스는 논리블록번호 상 인접한 bio 요청들을 하나의 request로 만들어 장치에서 한 번에 처리될 수 있도록 하기 위하여 플러그/언플러그 기능을 지원한다. 리눅스 커널 2.6.39 이전 버전에서는 bio 요청을 포함하는 request 구조체를 만들기 전에 플러그 안에 인접한 요청을 포

함하는 request 구조체가 있는지 확인하고, 이 경우 request 안에 bio 구조체를 포함시킨다. 또, request 큐가 비어 있는 경우에 입출력 요청은 플러그 리스트에 연결된다. 플러그에 연결된 요청들은 언플러그 처리되기 전까지는 디바이스에 요청될 수 없으며 플러그에 연결된 요청들은 기본적으로 3ms가 지나거나 4개 이상의 요청 만들어지면 언플러그 처리가 된다.

플러그 처리는 비작업 보장성 (non-work-conserving)을 갖기 때문에 리눅스 커널 버전 2.6.39부터 플러그 처리 방식이 크게 바뀌었다. 기존에 장치 별로 관리되던 플러그 방식이 쓰레드 별로 처리하는 방식으로 변경되었고, 플러그 처리를 하려면 프로그래머가 플러그 처리를 위한 코드를 명시적으로 호출해야 하도록 바뀌었다. blk_start_plug와 blk_finish_plug는 각각 플러그와 언플러그 처리를 위하여 프로그래머가 호출해야 하는 커널 함수이다.

리눅스는 버퍼캐시로 관리되는 블록의 입출력 요청을 위하여 파일시스템 상의 블록 크기 당 하나의 bio 구조체가 사용된다. 따라서 프리페쳐가 버퍼캐시로 관리되는 인접한 블록들을 하나의 request로 처리하려면 플러그 처리를 하여야 한다. 본 논문에서는 버퍼캐시로 처리되는 블록들이 병합된 경우에 하나의 request로 처리하기 위해서 플러그 처리를 이용하고 있다. 병합된 버퍼캐시 읽기 요청은 비동기적으로 장치에 요청될 수 있지만, 플러그를 사용하지 않으면 요청되는 블록의 개수만큼 request가 생성되기 때문에 명령 큐잉이 지원되지 않는 경우에는 입출력 처리 시간이 길어질 수 있다.

4.4 응용프로그램과 프리페쳐 동작의 병렬화

응용프로그램이 최초로 실행될 때 기동시퀀스를 수집하고, 저장 장치의 특성에 맞게 프리페칭될 순서 (프리페치시퀀스)를 최적화하여 디스크에 저장한다. 이 후, 같은 어플리케이션이 다시 수행되면 프리페쳐는 스케쥴된 기동시퀀스를 디스크캐시에 적재하는데, 이 때 프리페칭이 끝난 후 어플리케이션을 수행하는 동기 프리페칭 방식과, 프리페칭을 수행하는 쓰레드와 대상 어플리케이션을 동시에 수행하는 비동기 프리페칭 방식이 있다.

4.4.1 하드디스크를 사용하는 시스템

하드디스크를 사용하는 시스템에서는 어플리케이션 기동시간의 80~90%를 하드디스크 처리에 소비한다. 하드디스크의 처리 시간을 최적화하기 위해서는 헤드의 움직임을 줄이는 것이 중요하며, 이를 위해 기동시퀀스를 논리블록번호로 정렬하여 저장하였다. 디스크 프리페칭 작

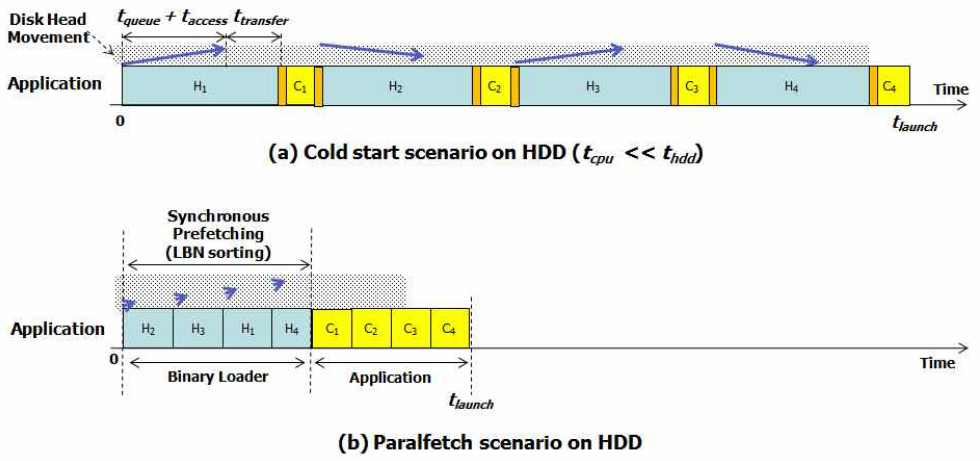

그림 16 하드디스크를 위한 동기 프리페칭

업이 응용프로그램 기동과 비동기적으로 수행되면 프로세서와 디스크가 동시에 활성화되는 효과가 있지만, 논리블록번호로 정렬된 프리페치 요청 사이에 응용프로그램이 발생시킨 디스크 요청이 끼어서 큰 디스크 헤드의 움직임이 발생할 가능성이 높아진다. 따라서 하드디스크 기반 시스템에서는 기동시간의 대부분을 차지하는 디스크 처리시간을 최적화하기 위해서 프리페칭이 완료될 때까지 대상 어플리케이션의 동작을 멈추어 놓는 동기프리페칭을 수행한다.

하드디스크 기반의 장치에서 프로세서와 디스크의 병렬적 사용을 통하여 제한적으로 기동시간을 단축시킬 수 있다. 그림 17은 동기 및 비동기 프리페치 방식이 결합된 하이브리드 프리페치 방식을 나타낸다. 하이브리드 모드로 수행 시에, 동기적으로 프리페칭되는 블록 중 페이지캐시로 관리되는 블록을 정해진 개수 만큼 비동기 프리페치에 추가하고 기동시간이 단축되는 경우에는 설정을 유지한다. 하이브리드 기법을 이용한 최적화는 하드디스크 및 하이브리드 하드디스크 등에 적용이 가능하다.

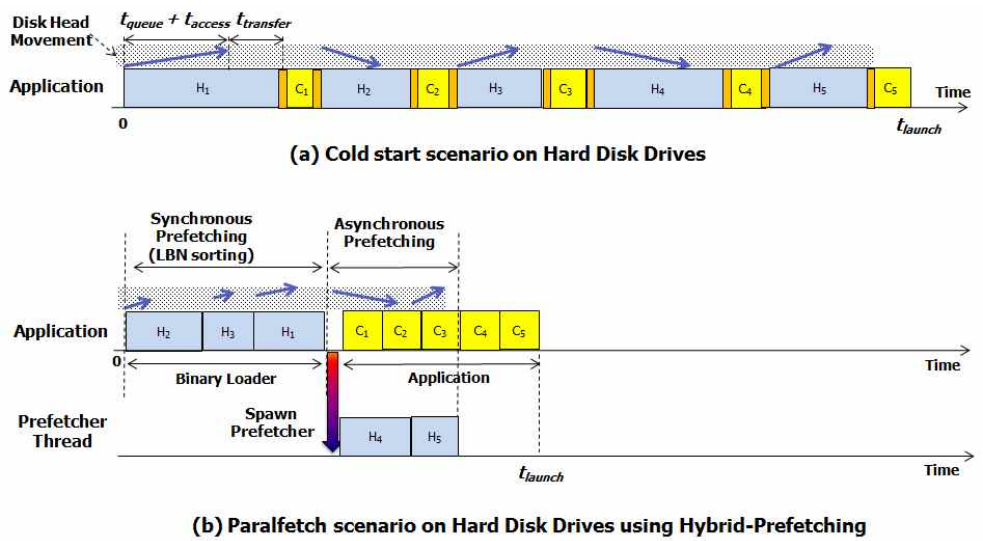

그림 17 하드디스크를 위한 동기, 비동기 하이브리드 프리페칭

4.4.2 SSD를 사용하는 시스템

플래시 기반의 저장장치는 기계적으로 움직이는 부분이 없기 때문에 요청 패턴에 상관없이 접근 시간이 거의 일정하다. 그래서 이 경우에는 논리블록번호로 정렬하는 것이 효과가 거의 없다. SSD 기반의 시스템에서는 그림 18-(c)와 같이 프리페칭을 수행하는 쓰레드와 대상 응용프로그램을 동시에 수행시켜서 프로세서와 디스크 자원을 동시에 활성화시켜 기동시간을 단축시키는 것이 중요하다. 프리페치시퀀스는 기동 시 수집된 블록의 접근 순서와 같게 유지시킬수록 좋다. 플래시 기반 시스템에서 사용되는 병합 기법은 병합 대상 블록 간의 요청 순서 상 거리를 제한하는 파라미터를 두어 최초에 수집된 순서를 가급적 유지한다.

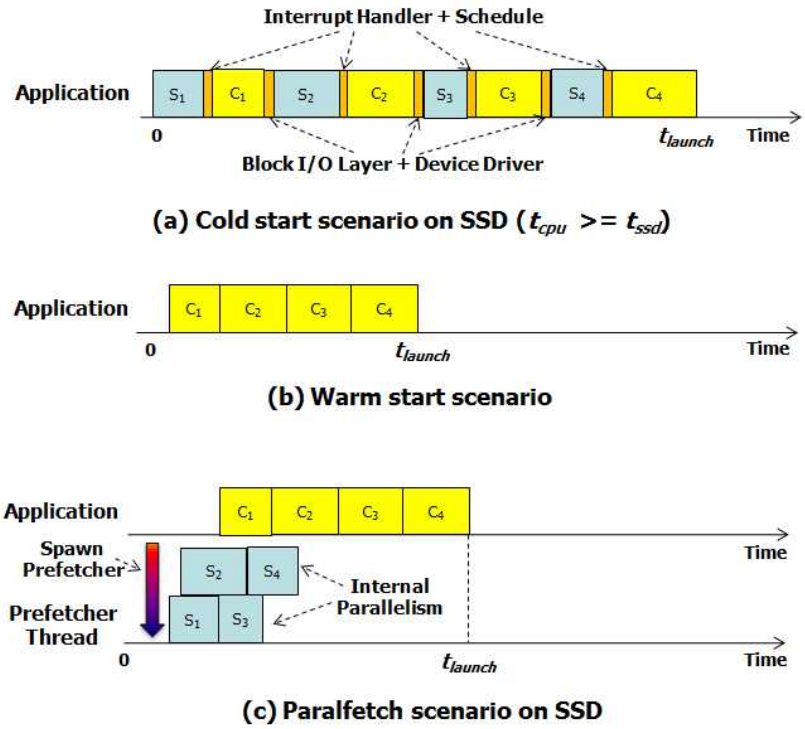

그림 18 SSD를 위한 비동기 프리페칭

4.4.3 다중 디스크를 사용하는 시스템

리눅스 커널에서는 컴퓨터에 장착된 디스크가 회전하는 매체를 포함하고 있는지 사용자에 알려주기 위하여 장치 별로 제공하는 rotational 프록 파일 (proc file)을 제공한다. 또, 디스크 장치들이 디바이스 사상자 (device mapper)로 관리되는 경우에는 내부 구성을 쉽게 알 수 있으며 디스크의 모델까지도 알 수 있다. 시스템에 서로 다른 종류의 여러 디스크가 장착되어 있는 경우에는 시스템 파티션 (예를 들어 루트 파티션) 용 디스크의 특성에 맞는 프리페치 방식을 적용한다. 여기에는 프리페치 최적화 방식, 그리고 프로세서와 디스크의 병렬적 사용 방법이 포함된다. 일반적으로 사용자는 빠른 디스크를 시스템 파티션을 위해 사용하는 경향이 있고, 일반적으로 응용프로그램 기동 시에는 유저 데이터가 저장되어 있는 파티션에서는 적은 수의 블록을 읽기 때문이다. 또, 여러 개의 물리 디스크로 구성되어 있는 경우, 프리페치 처리는 디스크 별 쓰레드로 수행된다.

4.5 기동시퀀스의 유효성 관리

수집된 기동시퀀스의 정확도는 프리페치의 성능에 큰 영향을 준다. 수집된 기동시퀀스에 다른 응용프로그램이 요청한 블록이 포함될 수 있고 블록이 업데이트 되면서 변경될 가능성이 있다. 라이브러리 또는 응용프로그램 업데이트, 그리고 새로운 프로그램의 설치는 기동 블록 변화의 주된 원인이다. 그 외에도 Matlab이나 Libreoffice는 프로그램 기동 시 사용자의 홈디렉토리를 검색하여 파일들을 조사하는데, 사용자의 홈디렉토리에 있는 파일들은 사용자의 시스템 사용에 따라 자주 변경된다.

표 4 기동시퀀스 유효성 관리를 위한 파라미터

종류	설명	기본 값
$T_{interval}$	기동시퀀스 유효성 검사 주기	3일
$Thr_{regeneration}$	기동시퀀스 재생성을 위한 부정확한 블록의 비율 임계치	10%

본 논문에서는 한 응용프로그램의 기동시퀀스에 다른 응용프로그램이 발생시킨 블록을 포함되어 있거나, 기동시퀀스 수집 이후 변경되는 블록들을 파악하고 프리페치시퀀스에 반영하기 위하여 주기적으로 변경사항을 조사한다. 응용프로그램 프리페쳐가 수행될 때 프리페치 대상 파일이 제거된 경우에 이를 로그로 남기고 유휴시간에 프리페치시퀀스 파일에서 삭제된 파일 항목을 제거한다. 또, 기동시퀀스 매니저를 정해진 주기 ($T_{interval}$) 마다 수행하여 프리페치 대상 파일의 수정시간이 프리페치 정보 파일의 생성 시간보다 늦은 경우에 이를 프리페쳐 정보 파일 (프리페치시퀀스 파일) 내의 헤더 영역 (uncertain_io_count와 uncertain_io_size)에 부정확한 입출력 요청의 수와 크기를 누적하고 두 값 중 하나라도 임계치 ($Thr_{regeneration}$)를 넘기면 프리페치 파일을 새로 생성하도록 하였다.

4.6 운영체제 부트 프리페쳐

운영체제 부팅시에 프리페칭이 수행되도록 커널 인자에 두 가지를 추가하였다. 첫째로 bootdev 인자는 운영체제 부팅에 사용되는 루트 디바이스 (최종적으로 '/' 디렉토리에 마운트되는 디바이스 파일 이름)의 이름을 전달하는 용도로 쓰인다. 둘째로 bootfetch 인자는 부트 페치의

단계를 알려 주는데 사용된다. 이 값이 3이면 부트 프리페치 정보 생성을 의미하고 1이면 비동기 프리페치, 1이면 동기 프리페치를 의미한다.

4.7 유휴시간 프리페쳐 인터페이스

유휴시간 프리페쳐는 가까운 미래에 수행될 응용프로그램을 과거의 사용 패턴을 통하여 예측하고, 사용자가 응용프로그램을 수행하기 전에 미리 해당 응용프로그램의 기동시퀀스를 프리페칭하는 기법이다. 응용프로그램의 사용 패턴을 분석하는 알고리즘은 기존에 연구된 것들이 많고 공개된 것도 있다. 이러한 알고리즘들을 본 논문에서 작성한 유휴시간 프리페치 인터페이스와 연동할 수도 있다. Paralfetch에서는 /proc/flashfetch/fetch_app 프록 파일을 통해서 유휴시간에 프로그램의 기동시퀀스를 프리페치 할 수 있다. 예를 들어 firefox의 기동시퀀스를 프리페치 하려면 터미널 상에서 "echo firefox > /proc/flash-fetch/fetch_app"을 타이핑하면 된다. 또는 프로그램 상에서 해당 프록 파일에 프리페칭할 어플리케이션의 이름을 기록하여도 된다.

4.8 실험 환경

본 논문에서는 제안한 프리페쳐의 성능을 평가하기 위하여 데스크탑 및 모바일 플랫폼에서 사용되는 운영체제와 응용프로그램을 이용하였다.

데스크탑 워크로드의 실험 환경

인텔 I7-2620m 2.8GHz 프로세서 (터보부스트 시 최대 3.4GHz)와 8GB의 주 메모리가 장착된 기기에 Fedora 16 64비트 버전을 설치하였다. 사용된 디스크의 종류는 시게이트 2.5인치 500GB 7200RPM 하드디스크와 인텔 320시리즈 80GB SSD이다.

파일시스템은 ext4를 사용하였고, SSD에서는 아이노드 미리읽기를 비활성화로 설정하였다. 하드디스크 기반의 저장장치에서 cfq 입출력 스케쥴러를 사용하였고, SSD 저장장치에서는 fiops를 사용하였다. 응용프로그램의 기동시퀀스를 수집하기 위한 시간값 (timeout)은 하드디스크 기반에서 30초, SSD에서 10초로 충분한 값을 주었다.

성능평가를 위한 벤치마크 (benchmark) 프로그램은 리눅스에서 많이 사용되는 14개의 프로그램을 이용하였다: Acrobat reader, Designer-qt 4, Eclipse, Firefox, Gimp, Houdini, Kdevelop, Konqueror, Labview, Libreoffice, Matlab, Scribus, Thunderbird, XilinxISE이다.

모바일 장치 워크로드의 실험 환경

Meego와 안드로이드 플랫폼 상에서 모바일 워크로드의 성능을 평가하였다. 먼저 Meego 환경에서의 실험은 인텔 Atom N470 1.86GHz 프로세서, 2GB의 주 메모리, 64GB의 샌디스크 SSD (모델 번호

SDSA3ED)가 장착된 태블릿 장치에 태블릿 용 Meego 1.2를 설치하여 제안한 프리페쳐의 성능을 실험하였다. 운영체제는 ext3로 포맷된 파티션에 설치하였고 아이노드 미리읽기를 비활성화로 설정하였다. 입출력 스케쥴러는 fiops를 사용하였고 응용프로그램의 기동시간을 감지하기 위한 시간은 20초로 설정하였다. 응용프로그램의 기동시간 측정을 위한 벤치마크 프로그램으로 Meego에서 많이 사용되는 6개의 프로그램을 이용하였다: Acrobat reader, Evolution, Firefox, Frozen-bubble, Ooffice-writer, Qt-creator.

안드로이드 플랫폼에서의 실험은 삼성 갤럭시 넥서스 폰에서 수행하였다. 갤럭시넥서스는 구글의 레퍼런스 폰 중 하나로 듀얼 코어 1.2GHz Cortex-A9 프로세서와 1GB의 메인메모리, 그리고 16GB의 내장 플래시가 장착된 스마트폰이다. 안드로이드 4.2.1 젤리빈 버전을 사용하였고 입출력 스케쥴러는 noop로 설정하였다. 또, 이 버전의 안드로이드는 기본적으로 ext4 파일시스템이 사용된다. 응용프로그램의 기동시간 감지를 위한 시간은 20초로 설정하였다. 벤치마크 프로그램으로 다음의 9개의 앱을 사용하였다: Camcard, Twitter, Office suite, Facebook, Kakaotalk, Naver, Aftermath, Anypang, Angrybird.

기동시간 측정 방법

응용프로그램은 윈도우 상에서 아이콘을 클릭하거나 터미널에서 명령어를 입력함으로써 기동이 시작된다. 기동 시작은 실행파일 로더가 수행되는 시간을 통하여 알 수 있다. 하지만 응용 프로그램의 기동이 끝나는 순간을 정확하게 분석하는 것은 쉽지 않다. 본 논문에서는 기동시간 측정을 위하여 성능평가에 사용된 프로그램을 하드디스크 기반 시스템에서 3회 수행하였고 blktrace를 이용하여 접근되는 블록 정보를 수집하였

다. 블록 정보 수집은 기동이 완료되었다고 판단되는 순간 해제하였다. 이 후, 세 개의 수집된 블록시퀀스에서 공통적으로 마지막에 접근된 정규파일의 이름과 블록 번호를 분석하였고 이 블록의 처리 완료 시점을 기동의 끝으로 판단했다. 모바일 플랫폼에서도 마찬가지 접근 방법을 적용하였다. 본 논문에서는 측정한 기동시간은 실행파일 로더의 수행 시작 시간과 기동 시 마지막으로 접근하는 정규파일의 블록이 디스크로부터 읽기 완료된 시점의 기간을 나타낸다.

기동시간 테스트를 위한 상황 설정 (scenario)

● 콜드스타트 (cold start): 디스크캐시 및 slab 캐시를 무효화한 후, 곧바로 응용프로그램을 실행하여 기동시간을 측정하였다. 캐시 무효화를 위하여 “sync; echo 3 > /proc/sys/vm/drop_caches“ 명령어를 터미널에서 수행하였다.

● 웜스타트 (warm start): 응용프로그램을 수행한 후, 기동 시 마지막으로 접근하는 정규파일 블록만 디스크캐시에서 무효화 시켰다. 이 후, 같은 프로그램을 한 번 더 실행하여 기동 시 마지막으로 접근하는 정규파일 블록의 처리가 완료될 때까지의 시간을 측정하였다.

● 구글-프리페치 (Google-Prefetch): 구글-프리페치의 동작을 재현한 프리페치 방식으로 프리페치시퀀스를 장치 번호, 아이노드 번호, 시작 블록 번호, 블록 길이를 이용하여 파일 수준에서 정렬한 방식이다. 구글 프리페치는 명시적으로 정규 파일만 프리페칭한다. 프리페치시퀀스에서 마지막으로 접근되는 정규 파일의 블록은 제거한 후 프로그램 기동 시 구글 프리페치 방식으로 기동시퀀스를 프리페칭하였다. 응용프로그램 실행 전에 디스크캐시와 slab 캐시는 무효화하였다.

● Paralfetch: 본 논문에서 제안한 프리페치 방식으로 4장에서 설명한 알고리즘에 따라 프리페칭하는 방식이다. 적용된 기법의 종류와 인자값은 결과 그래프에 설명하였다. 구글-프리페치 측정과 마찬가지로 마지막으로 접근되는 정규 파일의 블록은 프리페치시퀀스에서 제거하였고 프로그램 기동 전 디스크캐시와 slab 캐시는 무효화하였다.

4.9 응용프로그램 기동시간

● 기동시퀀스 수집

응용프로그램의 최초 기동을 통하여 수집된 기동시퀀스의 입출력 요청 개수와 전체 용량은 아이노드 미리읽기의 여부에 따라 다르다. 표 5와 6은 아이노드 미리읽기를 수행하지 않는 SSD와 미리읽기 값이 32로 설정된 하드디스크 환경에서 수집된 기동시퀀스의 읽기 요청 정보를 나타낸다.

표 5 수집된 기동시퀀스의 입출력 정보 (SSD)

응용프로그램	I/O 요청의 수	요청되는 블록의 총 크기 (KB)	접근하는 정규 파일의 수
Acrobat Reader	906	51,208	206
Designer-qt 4	2,157	54,472	327
Eclipse	4,764	97,460	522
Firefox	834	42,732	247
Gimp	1,807	51,480	843
Houdini	4,117	165,448	691
Kdevelop	3,601	126,012	2,042
Konqueror	1,811	41,532	336
Labview	2,123	79,416	468
Libreoffice	940	55,824	832
Matlab	4,457	153,008	710
Scribus	2,967	98,488	896
Thunderbird	749	65,048	269
XilinxISE	2,869	163,036	591

표 6 수집된 기동시퀀스의 입출력 정보 (HDD)

응용프로그램	I/O 요청의 수	논리블록번호 상 인접한 블록 병합 후 I/O 요청의 수	요청되는 블록의 총 크기 (KB)
Acrobat Reader	3,211	430	61,244
Designer-qt 4	6,089	957	68,468
Eclipse	7,598	1,654	114,136
Firefox	3842	505	53,168
Gimp	4,709	1,067	46,436
Houdini	6,565	2,024	174,740
Kdevelop	9,628	2,244	148,536
Konqueror	5,930	1,343	60,460
Labview	5,468	1,101	93,956
Libreoffice	3,347	513	63,736
Matlab	8,318	1,737	165,604
Scribus	6,727	1,358	99,168
Thunderbird	3,800	540	70,704
XilinxISE	5,356	952	171,052

하드디스크에서는 아이노드 미리읽기가 32블록으로 설정되어 있기 때문에 아이노드 블록 요청 시 실제로 32개의 블록이 동시에 요청된다. 32개의 요청은 각각 1개의 버퍼헤드를 사용하여 요청되는데, 리눅스 커널 2.6.39 부터는 기본적으로 플러깅 (plugging) 처리가 수행되지 않기 때문에 연속된 32개의 블록이 32개의 독립된 request 구조체를 통하여 비동기적으로 처리된다. 하지만 논리블록번호 상 연속적이기 때문에 하드디스크의 순차접근에 대한 처리 성능을 유도할 수 있다.

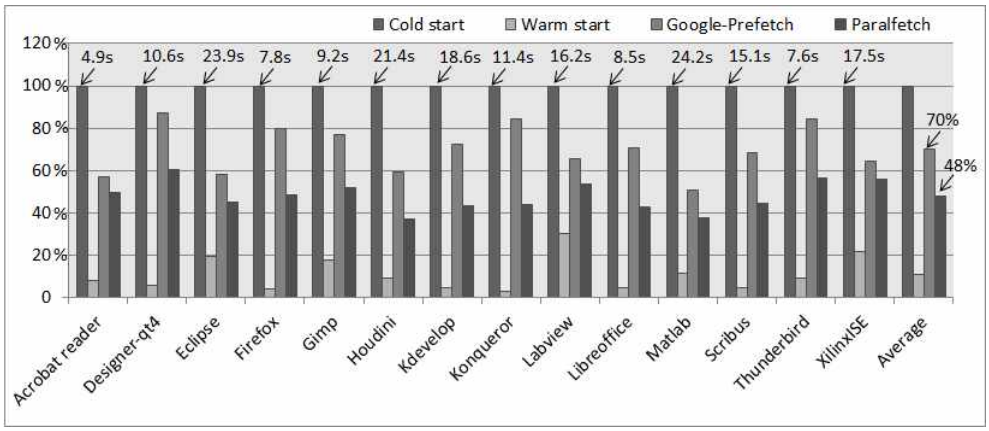

그림 19 응용프로그램의 기동시간

(HDD, 콜드스타트 시간에 정규화)

● 하드디스크를 사용하는 환경에서의 기동시간

그림 19는 하드디스크에서 콜드스타트, 웜스타트, 구글-프리페치, 그리고 제안한 프리페치 (Paralfetch)에서 벤치마크 프로그램의 기동에 소요되는 시간을 나타낸다. 하드디스크 환경에서는 디스크 시간이 기동시간의 대부분을 소비함을 알 수 있다. 또, 하드디스크의 경우 디스크 처리에 소비되는 시간의 편차가 발생한다. 블록 요청은 넌블록 모드(nonblock mode)로 요청이 되는데, 이 때 하드디스크 컨트롤러에서 명령 큐에 입력된 디스크 요청의 순서를 바꾸기 때문에 처리 시간의 편차가 발생한다. 편차를 줄이기 위하여 3번의 실행을 통하여 나온 평균 시간 값을 사용하였다.

구글-프리페치의 경우 버퍼캐시로 관리되는 아이노드, 디렉토리 블록 등을 명시적으로 프리페치하지 않고 파일 수준의 정렬을 사용하기 때문에 디스크 헤드의 움직임이 많아져서 기동시간이 제안한 방식에 비하여 22% 느리다. Paralfetch를 이용한 경우, 테스트 프로그램의 기동시간이

콜드스타트 시간에 비하여 52% 단축되었다.

● 하드디스크에서의 기동에 소비되는 에너지 분석

리눅스 커널은 ACPI (advanced configuration and power interface)를 통하여 배터리의 잔량을 알려주는 기능 및 관련 함수를 제공한다. 이를 이용하여 기동 과정에서 소비된 에너지를 측정하였다. 기동시간 측정과 마찬가지로 3회 측정한 값의 평균값을 사용하였다. 그림 20은 14개의 벤치마크 프로그램의 기동 시 소비된 에너지를 콜드스타트, 구글-프리페치, Paralfetch에 대해서 측정한 결과를 보여준다. 평균적으로 소비된 에너지는 소비 시간과 비율 상 거의 일치함을 알 수 있다.

● 하드디스크에서 빈공간 병합을 통한 기동시간 단축

빈공간 병합은 두 요청 사이의 빈 공간이 작을 경우, 이 공간을 포함하여 읽음으로써 입출력 요청의 수를 줄여서 프리페치 시간을 단축할 수 있다. 하지만 불필요한 공간을 디스크캐시에 읽어 메모리를 낭비하게 된다. 또, NCQ의 순서변경의 대상이 줄어들게 되어 성능이 낮아질 가능성

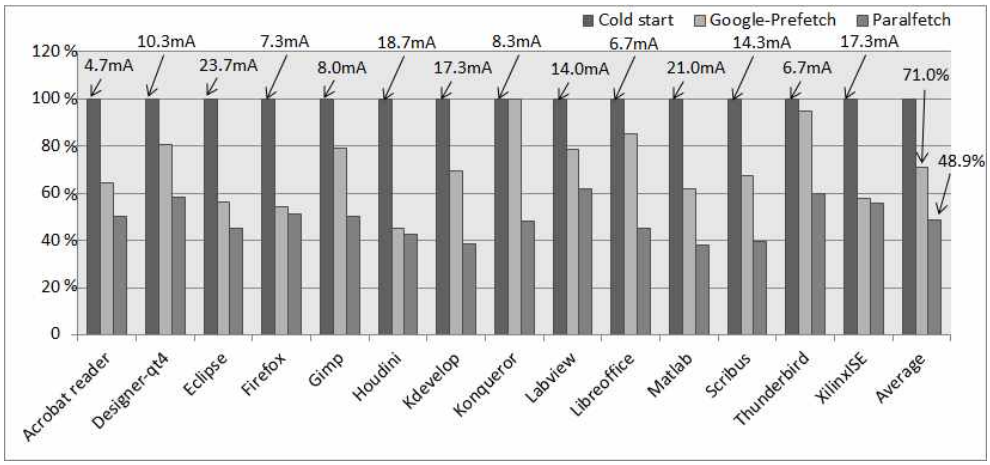

그림 20 응용프로그램 기동에 소비되는 에너지 분석

(HDD, 콜드스타트 에너지에 정규화)

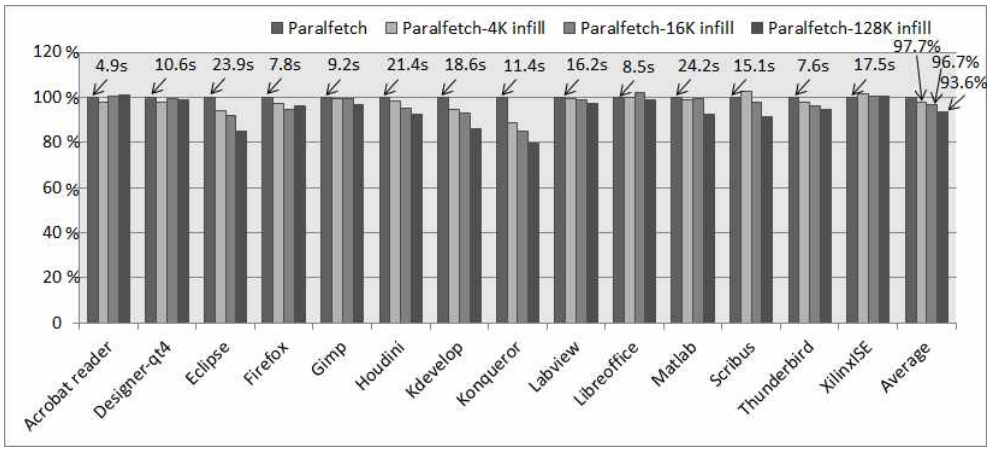

그림 21 빈공간 병합을 통한 기동시간 최적화

(HDD, 빈공간 병합을 하지 않는 Paralfetch 시간에 정규화)

도 배제할 수 없다. 그림 21은 인접한 블록만을 병합하는 Paralfetch의 기본 모드와 4K, 16K, 128K 이내의 빈공간 병합을 허용한 경우의 기동시간을 비교한 것이다. 평균적으로 큰 빈 공간을 허용할수록 기동시간이 단축되는 것을 볼 수 있다. 128KB 이하의 빈공간 병합을 허용한 경우, 기본 모드보다 기동시간이 6.4% 단축되었다.

표 7과 8은 각각 빈 공간 허용치를 늘렸을 때, 디스크 요청의 수와 프리페치 블록의 총 크기를 나타낸 표이다. 빈 공간 허용치를 늘리면 입출력 개수는 줄어들지만 기동시간은 약간만 개선된다. 또, 허용치를 128KB로 하였을 경우, 프리페치되는 블록의 전체 크기가 크게 증가함을 볼 수 있다. 빈공간 병합의 경우 입출력의 수도 중요하지만 요청이 병합되어 처리될 때의 처리될 섹터들의 물리적인 배치, 하드디스크 컨트롤러의 명령 순서 변경 능력 등에 따라 좌우된다. 또, 부가적인 블록의 프리페치로 인해 메모리가 낭비되기 때문에 사용에 주의를 해야 한다.

표 7 빈 공간 병합에서 빈공간 허용치 증가에 따른 입출력 수

응용프로그램	빈공간 허용치에 따른 I/O 요청의 수			
	기본 모드	4KB 허용	16KB 허용	128KB 허용
Acrobat Reader	430	417	397	341
Designer-qt 4	957	929	887	751
Eclipse	1,654	1,506	1,338	940
Firefox	505	492	466	377
Gimp	1,067	1,055	1,042	962
Houdini	2,024	1,839	1,468	967
Kdevelop	2,244	1,524	1,407	1,183
Konqueror	1,343	730	654	503
Labview	1,101	1,066	1,027	929
Libreoffice	513	497	478	406
Matlab	1,737	1,665	1,555	1,199
Scribus	1,358	1,328	1,267	1,106
Thunderbird	540	509	491	395
XilinxISE	952	925	888	780

하지만 4KB의 빈 공간 허용치를 적용하였을 경우에 낭비되는 메모리가 대부분의 응용프로그램에서 1MB 이하로 비교적 작았고, 기동시간이 평균 2.3% 단축되었다.

표 8 빈공간 병합에서 빈 공간 허용치 증가에 따른 입출력 총 크기

응용프로그램	빈공간 허용치에 따른 I/O 요청의 총 크기 (KB)			
	기본 모드	4KB 허용	16KB 허용	128KB 허용
Acrobat Reader	61,244	61,296	61,528	65,060
Designer-qt 4	68,468	68,580	69,048	78,012
Eclipse	114,136	114,720	116,652	137,812
Firefox	53,168	53,220	53,508	58,988
Gimp	46,436	46,484	46,628	51,828
Houdini	174,740	175,480	179,668	205,176
Kdevelop	148,536	151,416	152,692	167,428
Konqueror	60,460	62,912	63,768	73,220
Labview	93,956	94,096	94,532	100,604
Libreoffice	63,736	63,800	64,024	68,332
Matlab	165,604	165,892	167,128	189,020
Scribus	99,168	99,288	100,004	110,856
Thunderbird	70,704	70,828	71,032	76,776
XilinxISE	171,052	171,160	171,556	178,508

동기, 비동기 프리페칭 기법을 혼합한 하이브리드 프리페칭 방식을 이용하면 하드디스크 장치에서 응용프로그램의 기동시간을 추가적으로 단축할 수 있다. 동기적으로 프리페칭되는 블록 중 일부를 비동기적으로 프리페칭 함으로써 동기 프리페치에 소모되는 시간이 줄어서 전체 기동 시간이 단축될 수 있다. 14개의 테스트 프로그램 중 프로세서 시간이 가장 큰 5개에 대해서, 비동기적으로 프리페치 되는 블록의 수를 10~40%로 늘리면서 기동시간을 실험하였다. 또, 정규 파일 블록 중 늦게 요청되는 블록들을 비동기 처리 대상으로 선택하였다.

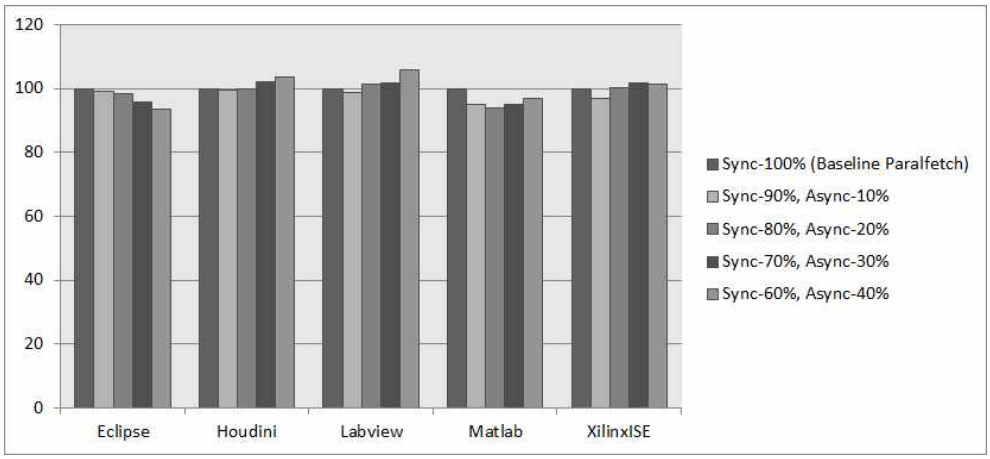

그림 22 동기, 비동기 하이브리드 프리페칭을 이용한 기동시간 단축
(Paralfetch 기본 모드의 기동시간에 정규화)

그림 22는 하이브리드 프리페칭의 실험 결과를 나타낸다. NCQ가 지원되는 하드디스크는 입출력 처리의 기아현상이 발생할 수 있고 [32], 이로 인해 응용프로그램과 프리페쳐의 동시 수행에 악영향을 줄 수 있다. 하지만 대부분의 블록을 동기적으로 프리페칭하면 동시성 문제가 어느 정도 해결되고, 비동기적으로 프리페칭되는 블록 처리 부분에서 디스크 헤드의 움직임이 NCQ의 명령 스케쥴링을 통하여 개선되어 기동시간이 단축될 수 있다. 하지만 비동기적으로 프리페칭되는 블록이 많으면 응용프로그램과 프리페칭의 동시 처리가 효과적으로 되지 않아서 오히려 느려지는 경우가 발생한다.

● SSD를 사용하는 환경에서의 기동시간

데스크탑이나 노트북에 사용되는 SSD는 많은 기능을 수행하는 컨트롤러를 장착하여 플래시 칩의 약점을 보완하고 있다. 성능적인 면에서는 여러 플래시 칩을 병렬적으로 사용하도록 분산하여 저장하고, 이를 병렬

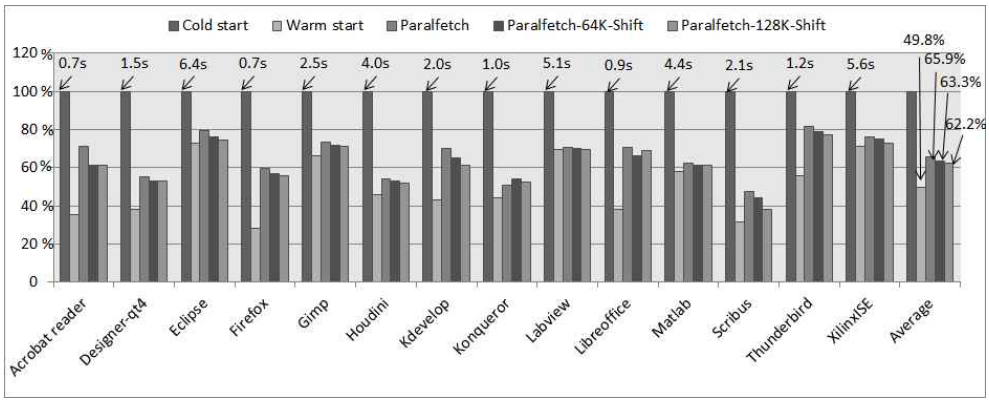

그림 23 응용프로그램의 기동시간
(SSD, 콜드스타트 시간에 정규화)

적으로 읽는다. 하지만 응용프로그램 기동 시 작은 요청들이 많이 발생하기 때문에 내부병렬성을 효과적으로 활용하기 어렵고, 동기적인 디스크 처리로 인하여 프로세서의 스케쥴링 오버헤드가 매 입출력 요청 시 발생한다. 성공적인 프리페칭은 디스크 처리 시간과 스케쥴 오버헤드 뿐 아니라 실제 디스크 입출력을 위해 입출력 요청을 생성하고 완료 처리를 하기 위한 프로세서 사용시간을 줄여준다. 그림 22는 콜드스타트, 웜스타트, Paralfetch 기본 모드, 그리고 Paralfetch의 메타데이터 쉬프트 크기를 64KB와 128KB로 설정한 경우의 기동시간을 보여준다.

SSD를 위한 Paralfetch의 기본 모드에서는 수집된 기동시퀀스의 익스텐트-의존성 분석을 수행하여, 놓친 아이노드 블록을 프리페치 목록에 추가한 후 프리페치 스케쥴을 종료한다. Paralfetch-64K-Shift는 Paralfetch 기본 모드에 메타데이터 쉬프트를 64KB 만큼 수행한 프리페치시퀀스를 이용하여 프리페칭하는 것을 의미한다. 이 경우 의존성이 있는 블록들 사이에 이들 블록과 의존성이 없는 블록을 64KB 추가 배치하여 프리페칭 시 내부병렬성의 효과적인 사용을 유도한다.

Paralfetch-128K-Shift도 쉬프트의 크기가 다르고 동작은 같다.

Paralfetch 기본 모드를 사용한 경우, 콜드스타트 시간 대비 34.1%의 기동시간이 단축되었고, 메타데이터 쉬프트를 64KB, 128KB를 수행한 경우, 콜드스타트 시간 대비 각각 36.7%, 37.8%의 기동시간이 단축되었다.

● SSD를 사용하는 Meego 플랫폼에서의 기동시간

Meego는 휴대폰, 태블릿 등을 위한 리눅스 기반의 운영체제 플랫폼으로 최근 Tizen 프로젝트로 흡수되었다. 모바일 장치는 일반적으로 낮은 소모 전력, 그리고 전력 대비 효율적인 성능을 추구하기 때문에 프로세서 및 저장장치의 성능에 제약이 있다. 이로 인해 모바일 플랫폼에서는 응용프로그램의 기동시간이 길고 디스크 시간이 큰 비중을 차지하기 때문에 실행시간 프리페쳐로 효과를 볼 수 있는 환경이다. 스마트폰이나 스마트패드 등에 사용되는 SSD는 현재 명령 큐잉을 지원하지 않기 때문에, 병합을 통하여 요청 크기를 크게하는 방법을 통하여 SSD의 내부병렬화를 극대화할 수 있다. 비동기 프리페칭 방식에서는 수집된 기동시퀀

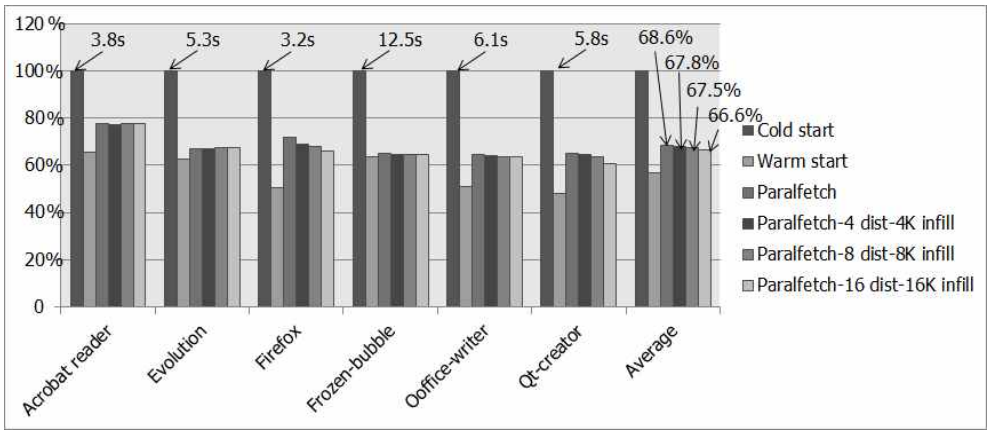

그림 24 Meego 용 응용프로그램의 기동시간

(콜드스타트 시간에 정규화)

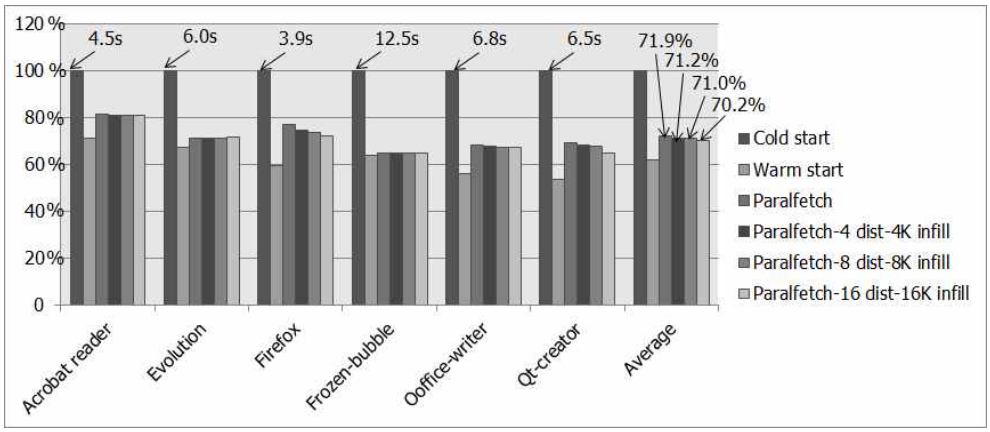

그림 25 Meego 용 응용프로그램의 기동시간
(화면 전환 적용, 콜드스타트 시간에 정규화)

스의 순서를 최대한 유지하는 것이 중요하기 때문에 블록의 기동시퀀스 내의 거리와 빈공간의 크기를 동시에 고려하여 병합해야 한다.

그림 24와 25는 Meego 환경에서 6개의 응용프로그램 기동 시 콜드스타트, 웜스타트, Paralfetch 기본 모드, 그리고 기본 모드에 거리기반 빈공간 병합을 적용한 경우의 기동시간을 보여준다. Meego 용 응용프로그램은 여러 개의 바탕화면을 가질 수 있어서 응용프로그램을 새로운 바탕화면에 기동시킬 수 있다. 새로운 바탕화면에 응용프로그램을 기동 시킬 때에는 추가적으로 화면 전환에 약 0.7초의 시간이 소요된다. 화면전환이 필요 없는 경우의 결과는 그림 24와 같고, 새로운 화면에 프로그램을 기동 시켜서 화면전환이 필요한 경우의 결과는 그림 25와 같다. Frozen-bubble의 경우에는 기동 중간에 화면 전환을 수행하기 때문에 기동시간이 두 실험에서 같다.

Paralfetch 기본 모드를 이용한 실험에서 화면전환 여부에 따라 콜드스타트 시간 대비 28.1~31.4%의 기동시간을 단축하였다. 입출력 거리

표 9 거리와 빈 공간 허용치에 따른 프리페치 요청의 크기

응용프로그램	거리와 빈공간 허용치에 따른 I/O 요청의 크기 (KB)			
	기본 모드	거리 4, 빈공간 4KB 허용	거리 8, 빈공간 8KB 허용	거리 16, 빈공간 16KB 허용
Acrobat reader	40,656	40,668	40,692	40,732
Evolution	36,328	36,364	36,432	36,568
Firefox	44,412	44,428	44,508	44,748
Frozen-bubble	32,224	32,232	32,244	32,336
Ooffice-writer	88,132	88,164	88,200	88,264
Qt-creater	94,280	94,344	94,500	95,184

표 10 거리와 빈 공간 허용치에 따른 프리페치 요청의 수

응용프로그램	거리와 빈공간 허용치에 따른 I/O 요청의 개수			
	기본 모드	거리 4, 빈공간 4KB 허용	거리 8, 빈공간 8KB 허용	거리 16, 빈공간 16KB 허용
Acrobat reader	801	753	733	695
Evolution	1,728	1,661	1,620	1,580
Firefox	789	751	731	699
Frozen-bubble	3,959	3,920	3,897	3,858
Ooffice-writer	1,692	1,539	1,500	1,456
Qt-creater	2,115	1,886	1,791	1,687

16이내의 16KB 이하의 빈공간을 병합하면 추가적으로 약 2% 정도의 기동시간 단축을 할 수 있다. 표 9와 10은 거리와 빈공간 허용치에 따른 입출력 요청의 개수와 총 크기를 나타낸다. Firefox를 제외하면 거리 기반 빈공간 병합으로 인한 메모리 낭비는 미미하였음을 볼 수 있다.

● 내장 플래시를 사용하는 안드로이드 환경에서의 기동시간

구글 안드로이드는 휴대폰 및 태블릿 환경에 가장 많이 쓰이는 모바일 운영체제이다. 리눅스 커널과 자바 기반의 프레임워크로 구성되어 있고, 아파치 라이선스를 따르기 때문에 안드로이드 소스를 이용하여 상업적인 소프트웨어 개발이 가능하다. 안드로이드는 리눅스 커널, 안드로이드 런타임, 라이브러리, 응용프로그램 프레임워크, 응용프로그램으로 구성되어 있고, 개발 언어는 기본적으로는 자바이지만 C나 C++ 로도 개발이 가능하다.

안드로이드는 전화, 메일, 문자, 캘린더, 위젯으로 관리되는 응용프로그램 (안드로이드 앱) 등은 가볍고 자주 사용되기 때문에 안드로이드 부팅 이후 초기화 프로세스에 의해 자동적으로 실행된다. 안드로이드는 새로운 안드로이드 앱을 실행시키기 위하여 커널의 실행파일 로더를 수행하지 않는다. 안드로이드에서는 zygote를 fork 하고, 생성된 자식 zygote 프로세스에서 새로 실행될 안드로이드 앱의 메인 클래스를 메모리에 로드함으로써 새로운 앱을 기동한다.

Paralfetch는 기본적으로 ELF 실행파일의 로더에서 프리페치 단계를 검사하지만 안드로이드에서는 zygote를 통하여 새로운 앱이 실행되기 때문에 zygote와 커널의 통신을 통하여 어떤 앱이 기동되는지 알려주고 커널에서 프리페치 단계를 처리하도록 하였다. “/proc/flashfetch/run_app” 프록 파일을 통하여 zygote는 새로 실행될 앱의 메인 클래스 이름을 알려준다. 커널은 클래스 이름과 pf 확장자가 합쳐진 이름의 프리페치 정보 파일이 존재하는 지 검사한 후, 파일이 존재하면 프리페치 단계, 파일이 없으면 프리페치시퀀스 생성 단계로 판단한다.

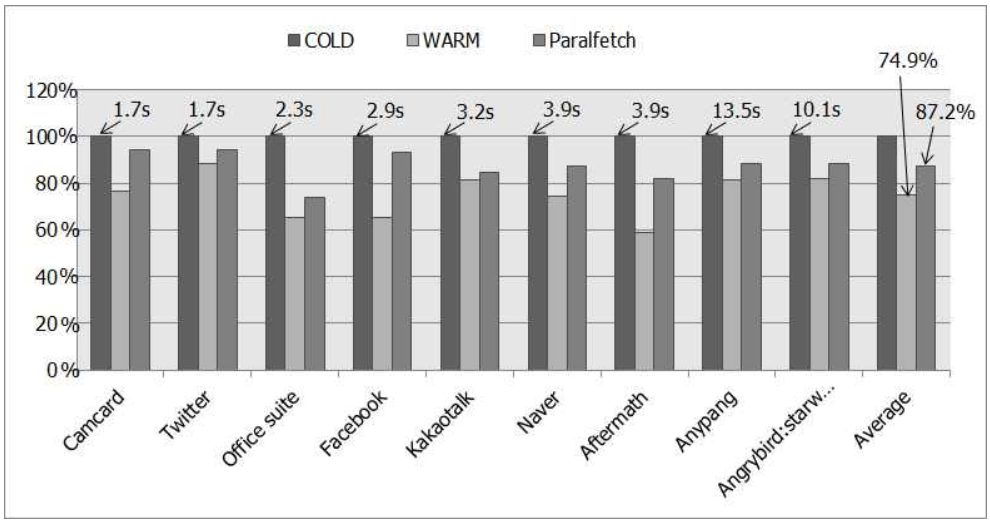

그림 26 안드로이드 용 응용프로그램의 기동시간

(콜드스타트 시간에 정규화)

그림 26은 벤치마크에 사용된 9개의 안드로이드 앱에 대한 기동시간을 보여준다. 삼성의 갤럭시 넥서스 폰에 Paralfetch가 적용된 안드로이드 4.2.1 젤리빈 버전을 설치하여 기동시간을 측정하였다. 안드로이드 플랫폼 용 응용프로그램들은 대체적으로 디스크시간이 큰 비중을 차지하지 않은 것을 알 수 있으며, Paralfetch를 이용하였을 경우 12.8%의 기동시간이 단축되었다.

4.10 운영 및 저장 공간 오버헤드

프리페쳐의 사용으로 인해 발생하는 오버헤드는 기법의 실질적(practical) 사용 가능성을 결정하는 중요한 요소이다. 본 논문에서 제안한 기법의 운영 및 저장 공간 오버헤드를 프리페쳐 사용 응용프로그램과 프리페쳐 미사용 응용프로그램으로 나누어 설명하겠다.

Paralfetch가 활성화된 경우, 프리페쳐를 사용하지 않는 프로그램에 부과되는 오버헤드는 프로그램 기동 시 실행파일 로더에서 프리페치 단계를 검사하는 것과 디스크캐시 미스 (miss)로 인하여 블록 레이어에 입출력 요청을 할 때, 요청을 로그로 남겨야 하는지 판단하기 위하여 로깅 여부를 확인하는 변수 검사 오버헤드가 있다. 이는 측정이 어려울 정도로 낮은 오버헤드이다. 구글-프리페치의 경우 프리페치 단계를 검사하기 위하여 실행파일의 이름과 몇 가지 정보를 이용하여 해시 값을 계산하고 블랙리스트와 화이트리스트에 있는지 검색하는 오버헤드를 갖는다. 또, 기동시퀀스 수집을 위하여 페이지캐시로부터 방출되는 페이지 마다 로그를 남겨야할지 검사해야하기 때문에 Paralfetch보다 오버헤드가 크다.

Paralfetch는 작은 디스크 공간을 이용하여 프리페치 정보를 관리한다. 24바이트의 헤더와 프리페치 블록 요청 당 64비트 환경에서는 24바이트, 32비트 환경에서는 20바이트의 정보 저장 공간을 요구한다. 윈도우즈 프리페쳐의 경우 프리페치 정보를 저장하기 위하여 응용프로그램마다 수 백 킬로바이트에서 수 메가바이트 수준의 디스크 공간을 요구한다. 구글-프리페치는 페이지캐시로 접근되는 요청들에 대해서 Paralfetch와 같은 수준의 공간을 사용한다. 표 11은 응용프로그램의 프리페치 정보를 저장하기 위하여 Paralfeth가 사용하는 디스크 공간을 나타낸다. 응용프로그램에 따라 수집되는 요청 블록의 개수가 다르고 디스

크 종류에 따라 병합의 정도가 다르다. SSD에서 14개 응용프로그램의 프리페치 정보 파일의 크기를 모두 합하여도 819KB로 매우 작다. 하드디스크의 경우 인접한 블록의 병합 횟수가 많아 프리페치 정보파일의 크기가 작다.

하드디스크와 같이 동기적인 방식으로 프리페칭을 수행하는 경우, 윔스타트에서는 동기적인 프리페칭 부분이 오버헤드가 된다. 이 부분에 대한 오버헤드는 위의 14개의 응용프로그램 중 Libreoffice에서 3ms로 가장 작았고 Matlab에서 6ms로 가장 컸다. 또, SSD를 사용하는 환경에서

표 11 64비트 데스크탑 워크로드의 프리페치 정보 파일의 크기

응용프로그램	SSD에서 프리페치 정보 파일의 크기 (Byte)	HDD에서 프리페치 정보 파일의 크기 (Byte)
Acrobat Reader	21,768	10,344
Designer-qt 4	51,792	22,992
Eclipse	114,360	39,720
Firefox	20,040	12,144
Gimp	43,392	25,632
Houdini	98,832	48,600
Kdevelop	86,448	53,880
Konqueror	43,488	32,256
Labview	50,976	26,448
Libreoffice	22,584	12,336
Matlab	106,992	41,712
Scribus	71,232	32,616
Thunderbird	18,000	12,984
XilinxISE	68,880	22,872

비동기 프리페칭 방식을 사용한 경우 프리페치를 사용한 경우와 프리페치를 사용하지 않은 경우의 웜스타트 시간은 차이를 측정하기 어려울 정도로 작았다. 프리페치시퀀스는 수집된 접근 블록 정보를 최적화하여 생성하는데 여기에 소비되는 시간은 실험에 사용된 모든 실험에서 500ms 이하였다.

4.11 커널 수준 프리페쳐의 안전성

커널 수준의 프리페쳐는 버퍼캐시 및 페이지캐시에 기동 블록을 읽어놓기 위하여 __breadahead와 force_page_cache_readahead 함수를 호출한다. 두 함수에 전달되는 인자 중 다른 쓰레드와 경쟁이 발생할 수 있는 것은 참조카운터를 증가시켜 메모리로부터 해제되지 않도록 하였다. 두 함수의 동작 과정에서 발생하는 동기화 문제는 커널에서 처리해 주기 때문에 프리페쳐를 사용함으로 인해 추가적으로 발생하는 문제는 없다. 하지만 페이지캐시에 디스크 블록을 캐싱할 때, 파일 수준의 정보가 아닌 아이노드 구조체의 i_mapping 멤버를 전달하기 때문에 비효율성이 발생할 수 있다. 예를 들면, 어떤 프로그램이 A라는 파일의 첫 블록을 기동 시에 항상 읽는 경우, 이 파일이 재생성되어 새로운 아이노드 번호를 할당 받았다고 가정해 보자. 이 상황에서 파일 이름과 블록 번호로 프리페칭을 하면 아이노드 번호가 변경되어도 A 파일의 첫 번째 블록이 프리페칭 될 것이다. 하지만, 커널 수준의 프리페쳐처럼 정규 파일의 프리페칭을 위하여 아이노드 수준의 정보를 사용하여 프리페칭하는 경우에는 A 파일의 아이노드가 변경되었기 때문에, 프리페칭되는 아이노드가 유효한 경우, 다른 파일의 내용을 프리페칭 하는 결과가 되어 비효율성이 발생할 수 있고, 아이노드가 삭제된 경우에는 프리페칭에 실패

하게 된다. 전자의 경우, A가 아닌 다른 파일을 프리페칭한 결과가 되고 A 파일의 내용을 잘못 프리페칭한 것이 아니기 때문에 안전하다.

효율성과 안전성은 다음과 같이 정의될 수 있다.

- 효율성: 프리페쳐가 미리읽어 놓은 내용을 응용프로그램이 읽어가는 지 여부

- 안전성: 응용프로그램이 프리페쳐의 동작으로 인하여 다른 데이터 값을 읽어가는 지 여부

본 논문에서 사용한 커널 수준의 프리페치 방법이 안전하다는 것을 다음의 세 가지 경우로 분류하여 설명하겠다. 기존의 커널 코드는 안전하여 아이노드 하부 처리는 안전하다고 가정한다. 또, 기동에 사용되는 파일 (기동 파일)은 1개이며 아이노드 번호는 100번으로 가정하고 설명하겠다.

● 프리페쳐 동작 중 기동 파일이 제거되지 않으며, 기동 정보 수집 때와 아이노드 번호가 같은 경우: 기동 시와 아이노드 번호가 같고 프리페치 중에 아이노드가 변경되지도 않는 경우이다. 아이노드 하위 처리는 커널 수준에서 동기화를 해 주기 때문에 안전하다.

● 프리페쳐 동작 중 기동 파일이 제거되지 않으며, 기동 정보 수집 때와 아이노드 번호가 다른 경우: 기동 시 수집된 아이노드 번호에 해당하는 정보가 파일 삭제로 인하여 디스크에서 제거된 경우이거나, 파일이 삭제된 후 다른 파일이 해당 아이노드를 사용하는 경우이다. 전자의 경우, 프리페치에 실패하기 때문에 시스템에는 아무런 영향이 없다. 후자의 경우, 프리페쳐가 다른 파일의 내용을 프리페칭한 결과가 된다. 파일

의 내용 뿐 아니라 아이노드도 이미 다른 파일이 사용하는 것이기 때문에 안전하다. 하지만 응용프로그램이 해당 블록을 읽어가지 않으면 일시적인 메모리 낭비가 될 수 있다.

● 프리페쳐의 동작 중 파일이 제거된 경우: 프리페쳐가 100번 아이노드와 파일의 내용을 프리페칭하는 중에 다른 쓰레드에 의하여 100번 아이노드에 해당하는 경로명 (path name)이 제거된 경우이다. 이 상황에서 100번 아이노드에 해당하는 파일의 경로와 관련 디스크 블록들은 디스크 상에서 모두 해제가 된다. 또, 경로명과 아이노드의 연결은 끊어지게 되어 응용프로그램이 경로 탐색을 통하여 접근할 수 없는 아이노드가 된다. 또, 100번 아이노드는 메모리 상에서는 유효하기 때문에 다른 파일에 해당 아이노드 번호가 할당되지 않는다. 결국, 100번 아이노드는 메모리 상에서 해제되기 전까지는 응용프로그램이 접근할 수 없기 때문에 안전하다. 하지만 메모리에서 100번 아이노드가 회수될 때 까지 해당 메모리 공간이 낭비되는 비효율성이 발생할 수 있다.

제 5 장 유저 수준 실행시간 프리페처의 설계, 구현 및 평가

5.1 유저 수준 프리페처의 소개 및 구조

디스크 블록을 디스크캐시에 적재하는 것은 커널 영역 뿐 아니라 유저 영역에서도 가능하다. 응용프로그램이 기동될 때, 요청되는 대부분의 블록들은 응용프로그램으로부터 요청되고 이는 디스크캐시에 적재된다. 따라서 접근되는 블록들의 정보를 수집하고 적절한 시스템 콜을 호출하여 직접 또는 간접적으로 해당 블록들을 디스크캐시에 적재하도록 유도함으로써 프리페처의 기능을 수행할 수 있다. 리눅스에서는 blktrace

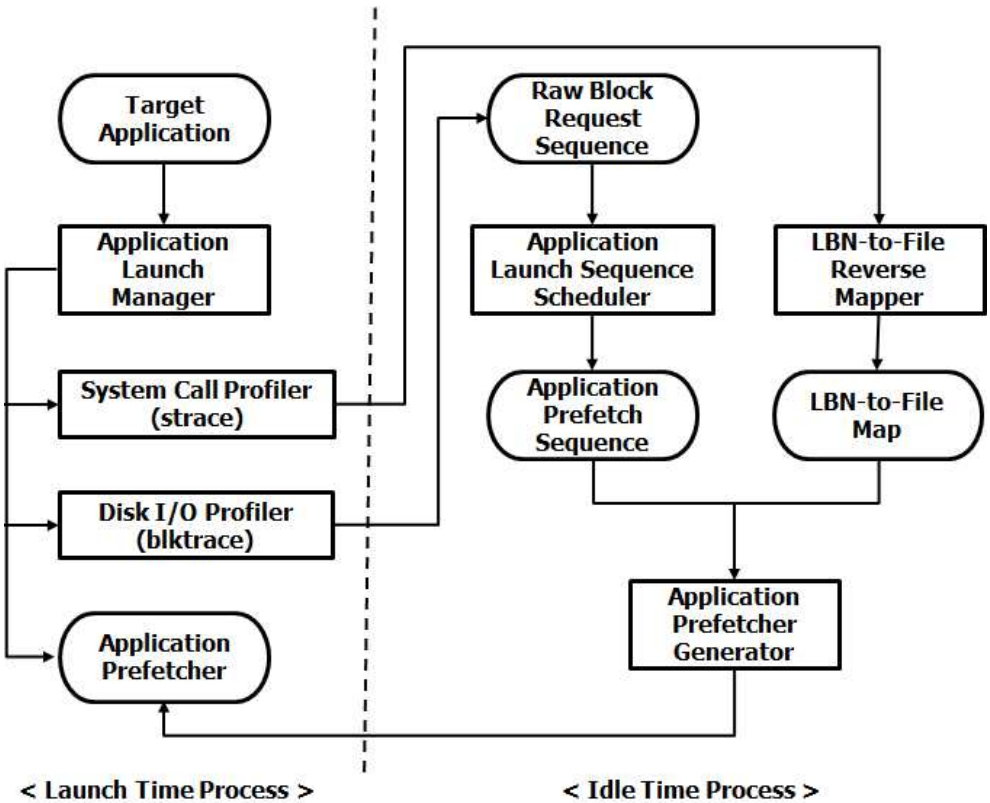

그림 27 FAST의 프레임워크

[38]를 이용하여 블록 수준의 입출력 정보를 추출 할 수 있다.

본 논문에서 제안한 유저 수준의 프리페처인 FAST (Fast Application STarter)의 구조는 그림 26과 같다. 응용프로그램의 기동 시 발생되는 블록 수준의 접근 정보를 blktrace를 이용하여 수집한다. 수집된 블록들을 페이지캐시에 적재하기 위해서는 해당 블록을 캐싱할 수 있는 적절한 시스템 콜을 호출하여야 한다. 하지만 시스템 콜은 블록 수준의 정보가 아닌 파일 수준의 정보를 요구한다. 따라서 블록 수준의 정보를 파일 수준의 정보로 변환하기 위한 역사상 (reverse mapping)을 수행하여야 한다. 결국, 블록 수준의 접근 정보는 블록-파일 역사상을 통하여 이를 디스크캐시에 적재할 수 있는 시스템 콜을 호출하는 코드로 변환되고, 컴파일러를 통하여 프리페칭을 수행하는 프로그램이 만들어진다. 만들어진 프리페쳐 프로그램을 대상 프로그램과 동기적 또는 비동기적으로 실행함으로써 대상 프로그램의 기동시간을 단축할 수 있다. 블록-파일 역사상을 위해서는 파일시스템의 분석 과정이 요구되는데, 본 논문에서는 ext3 파일시스템을 대상으로 구현하였다.

유저 수준의 프리페쳐는 커널 컴파일을 할 필요가 없기 때문에 커널 소스가 공개되지 않은 운영체제에도 적용할 수 있다는 장점이 있다. 하지만, 프리페칭을 수행하려고 하는 운영체제 환경에서 디스크의 접근 정보를 추출 할 수 있는 방법이 제공되어야한다 [38][39]40]. 예를 들어, 아이폰이나 아이패드에 사용되는 IOS 운영체제의 경우, 디스크 블록 접근정보를 수집할 수 있는 도구가 제공되지 않기 때문에 다른 방법으로 시퀀스를 수집해야한다. 또, 운영체제마다 디스크캐시를 관리하는 기법이 다르기 때문에 파일시스템 분석을 통하여 기동에 사용되는 블록을 디스크캐시에 적재하는 방법에 대한 분석이 필요하다.

5.2 응용프로그램의 프리페치시퀀스 생성

5.2.1 디스크 입출력 정보 수집

리눅스는 블록 계층에서 디스크 접근 정보를 감시하고 이를 유저 영역에 전달하기 위한 blktrace 도구를 제공한다. 블록 계층에서 입출력 요청, 병합, 플러그/언플러그, 다른 디바이스로 사상, 입출력 스케쥴러로 전달, 디바이스 드라이버로 전달, 입출력 처리 완료 등 다양한 이벤트를 수행할 때 해당 블록 접근 정보가 생성되며 접근 정보는 시간, 프로세스 ID, 이벤트 종류, 장치 번호, 섹터 번호, 요청 크기 등을 포함한다. 디스크캐시 무효화 상태에서 기동시퀀스를 수집하기 위해서 먼저 slab과 페이지캐시를 무효화 시키는 “sync; echo 3 > /proc/sys/vm/drop_caches” 명령을 수행한다. 그리고 blktrace를 이용하여 시스템에 장착된 디스크 장치에서 발생하는 <장치 번호, 장치 내 섹터 번호, 입출력할 섹터의 수>를 수집하고, 수집된 정보 중에 읽기 요청이면서 디바이스에 드라이버에 요청되는 정보만 추출한다.

5.2.2 기동시퀀스 추출

대상 프로그램과 관련이 없는 프로세스로부터 발생된 요청정보가 기동시퀀스에 포함되어 있을 수 있기 때문에 이를 제거하기 위하여 blktrace를 이용하여 수집한 2개의 기동시퀀스에서 공통적으로 발견되는 블록들을 프리페치시퀀스로 추출한다. drop_caches 프록 파일을 이용한 디스크캐시 무효화 방법은 slab 할당자로 관리되는 아이노드나 디렉토리 엔트리를 완벽하게 무효화시키지 못하기 때문에 여러 기동시퀀스에서 공통 블록을 추출할 경우에는 아이노드나 디렉토리 엔트리 정보와 같이

slab 할당자를 통하여 이중 캐싱되는 블록들의 정보를 잃어버릴 수 있다.

5.2.3 프리페치시퀀스 스케쥴

기동시퀀스를 빠르게 프리페칭하기 위해서 디스크의 특성에 따른 블록 요청 순서 변경과 병합 기법을 이용하여 프리페치시퀀스를 최적화하여야 한다. 디스크의 기본적인 특성을 활용하기 위하여 플래시 기반의 장치에서는 수집된 기동 순서를 변경하지 않지만 하드디스크와 같이 접근 시간 최적화가 필요한 경우에는 기동 중 수집된 블록 시퀀스를 논리 블록번호 순으로 정렬한다. 또, 여러 개의 디스크로 구성된 경우 디스크별로 프리페칭이 수행될 수 있도록 장치 번호에 따라 기동시퀀스를 분리한다.

5.3 블록-파일 사상 (Map)

5.3.1 블록-파일 사상의 소개

유저 수준의 구현에서 디스크캐시에 기동시퀀스를 적재하기 위해서는 운영체제에서 제공하는 시스템 콜을 이용하여야 한다. 리눅스에서 제공되는 디스크 입출력 관련 시스템 콜은 파일 수준의 정보를 인자로 요구한다. 따라서 응용프로그램 기동 중 수집된 블록 수준의 정보를 디스크캐시에 적재하기 위한 시스템 콜의 파일 수준의 인자로 변환하여야 한다. 하지만 대부분의 파일시스템에서 블록 정보를 파일 수준의 정보로 변환하는 기능을 제공하지 않는다. 하지만 파일 수준의 정보를 블록 수

준으로 변환하는 것은 어렵지 않다. 정규 파일에 대한 블록 정보 변환은 ioctl 시스템 콜에 FIEMAP (또는 FIBMAP)을 명령 인자로 주어 추출이 가능하다. 하지만 이 방법에서 아이노드 블록이나 디렉토리 파일에 대한 블록 정보는 제공하지 않는다. 따라서 정규 파일의 데이터 뿐 아니라 그 외의 모든 블록의 사상 정보 수집을 위해서는 파일시스템 수준의 분석을 수행해야 한다.

본 논문에서는 블록-파일 역사상을 위하여 역사상을 수행할 파일들의 이름 목록을 인자로 받고, 해당 파일들에 대한 블록-파일 사상 정보를 생성하였다. 블록-파일 사상은 검색의 속도를 높이기 위하여 빨강검정 트리 (red-black tree)를 이용하여 관리한다. 트리의 각 노드는 탐색을 위한 키로 논리블록번호가 사용되고, 노드의 데이터에는 해당 블록의 종류와 종류에 따른 부가 정보가 포함된다. 데이터의 종류에는 슈퍼 블록, 그룹 디스크립터, 아이노드 블록 비트맵, 데이터 블록 비트맵, 아이노드 테이블, 그리고 데이터 블록이 있으나 유저 수준에서 명시적으로 디스크 캐싱이 가능한 아이노드 블록, 디렉토리 엔트리 블록, 데이터 블록을 프리페칭의 대상으로 하였다. 노드 데이터의 종류가 아이노드 블록인 경우 해당 블록이 저장하고 있는 아이노드 번호의 범위를 기록한다. 또, 노드의 데이터 종류가 데이터 블록인 경우에는 정규 파일, 디렉토리 파일, 심볼릭 링크 파일로 구분하고 정규 파일인 경우에는 장치 번호, 아이노드 번호, 파일 내 블록 번호 등을 기록한다. 하지만 디렉토리 파일이나 심볼릭 링크 파일의 데이터 블록인 경우에는 장치 번호와 아이노드 번호만 노드의 데이터로 기록한다. 심볼릭 링크 파일의 이름이 60자 이하인 경우에는 아이노드에 파일 이름이 저장되기 때문에 블록-파일 사상 정보를 생성하지 않아도 된다.

5.3.2 기동시퀀스 관련 파일 목록의 수집

리눅스 시스템에는 보통 수만 개 이상의 파일이 저장되어 있기 때문에 모든 파일에 대해서 블록-파일 사상 정보를 생성하는 것은 현실적으로 불가능하다. 가장 큰 이유는 사상에 요구되는 계산 및 디스크 처리 오버헤드가 비현실적으로 크고, 사용자가 시스템을 사용하면서 파일이 생성, 삭제, 변경되면 사상 정보가 바뀌기 때문이다. 다행히도 대부분의 응용프로그램은 기동 시, 수백 개의 파일만을 접근한다. 본 논문에서는 프리페쳐 프로그램을 생성 할 때, 파일시스템 분석을 통하여 프로그램 기동과 관련된 파일들에 대해서 블록-파일 사상 정보를 생성하고 이를 이용하여 프리페쳐 프로그램 코드를 생성하는 방법을 사용하였다.

일반 유저의 권한을 갖는 유저 수준 코드에서 디스크에 엑세스를 하기 위해서는 시스템 콜을 이용하여야 한다. 프로세서 아키텍쳐에서 지원하는 다른 방법들은 운영체제에 의해서 막혀 있다. 응용프로그램 기동 시 호출되는 시스템 콜을 수집하기 위하여 strace 툴을 이용하였고, 기동 시 사용되는 파일의 경로 (pathname)를 수집하기 위하여 파일의 경로를 시스템 콜의 인자로 요구하는 모든 시스템 콜에서 파일의 경로에 해당하는 값을 추출하였다. 하지만, 프로그램 기동 시 접근되는 모든 파일이 해당 프로그램의 기동 프로세스에 의하여 열리거나 접근되는 것은 아니다. X-Windows 상에서 동작하는 프로그램들은 X-Windows 관련 프로세스를 통하여 파일이 열리거나 디스크 엑세스를 발생시키기도 한다. 기동 시 발생되는 약 10~20%의 블록 접근은 X-Windows 관련 프로세스가 열어놓은 파일에서 발생하였다. X-Windows가 열어놓은 파일들을 블록-파일 사상에 이용하기 위해서 lsof 명령을 수행하여 얻은 파일 목록을 블록-파일 사상을 위한 파일 목록에 추가하였다. lsof 명령은

proc 파일시스템을 통하여 현재 수행 중인 모든 프로세스가 열어 놓은 파일들의 정보를 알려준다.

5.4 유저 수준의 프리페쳐 프로그램 생성

유저 수준의 프리페쳐 프로그램은 대상 응용프로그램의 기동시퀀스를 디스크캐시에 적재하는 프로그램이다. blktrace를 이용하여 수집된 기동시퀀스는 기동시퀀스 스케쥴러를 통하여 디스크의 특성에 맞게 블록 요청 순서를 변경한다. 이러한 과정을 거쳐 만들어진 프리페치시퀀스에서 블록 수준의 접근 정보와 프로그램 관련 파일의 블록-파일 사상 정보를 이용하여 프리페쳐 코드가 생성되고 컴파일러를 통하여 프리페쳐 실행파일로 만들어진다.

표 12는 블록 종류에 따라 디스크캐시에 적재하기 위한 시스템 콜을 나타낸다. 블록의 종류가 아이노드 테이블 블록인 경우 해당 블록이 저장하고 있는 아이노드의 범위를 블록-파일 사상을 통하여 알 수 있다. 해당 아이노드 범위 안의 파일을 프로그램 기동 시 접근할 경우 관련 파일들을 open 시스템 콜을 이용하여 모두 연다. 만약에 파일의 데이터를 추후에 접근하지 않는다면 lstat 시스템 콜을 이용하여 관련 블록을 캐

표 12 블록 종류에 따른 블록 캐싱 시스템 콜

블록 종류	시스템 콜
아이노드 테이블 블록	open() 또는 lstat()
데이터 블록: 디렉토리	opendir()와 readdir()
데이터 블록: 정규 파일	posix_fadvise()
데이터 블록: 심볼릭 링크 파일	readlink()

싱할 수 있다. 데이터 블록의 경우, 세부 용도에 따라 적절한 시스템 콜을 호출한다. 예를 들어, 디렉토리 파일의 데이터 블록을 디스크에 캐싱해야하는 경우에는 opendir과 readdir 시스템 콜을 이용하여 디스크캐시에 적재한다. opendir 및 readdir은 동기적으로 관련 블록들을 처리하기 때문에 비동기적 처리에 비해 명령 큐잉의 이점을 이용하지 못한다. 또, 분석 복잡도를 낮추기 위해서 원하는 디렉토리 데이터 블록만 읽는 대신 디렉토리 파일의 모든 블록을 읽는다. 정규 파일의 데이터 블록은 posix_fadvise 시스템 콜에 POSIX_FADV_WILLNEED 인자를 이용하여 비동기적으로 프리페칭을 수행하고, 심볼릭 링크 파일의 데이터 블록은 readlink를 호출하여 디스크캐시에 해당 블록을 적재한다.

Ext3 파일시스템의 경우, 정규 파일의 최초 12개 블록의 위치는 아이노드에 저장된다. 파일이 그보다 클 경우, 그 크기에 따라 아이노드의 단일 간접 블록 포인터, 이중 간접 블록 포인터, 삼중 간접 블록 포인터와 데이터 블록을 이용하여 실제 데이터가 저장된 위치를 기록한다. 간접 블록 포인터를 통하여 저장되는 블록 위치 저장 블록들은 유저 수준에서 명시적으로 프리페칭이 불가능하다. 하지만, 정규 파일의 데이터 블록을 접근할 때 관련된 위치를 기록한 간접 포인터 블록들은 자동적으로 접근되어 디스크캐시에 적재된다.

5.5 응용프로그램 기동 관리자

응용프로그램 기동 관리자는 응용프로그램의 기동을 감지하고 적절한 동작을 수행하는 역할을 한다. 본 논문에서는 execve 라이브러리 래핑 (wrapping)을 통하여 응용프로그램의 기동을 감지하였다. 응용프로그램 기동 관리자는 1) 기동시퀀스 수집, 2) 응용프로그램 프리페쳐 생성, 3) 프리페쳐 수행의 세 단계로 동작한다. 응용프로그램 기동 관리자는

여러 가지 설정 값과 상태 값에 따라 수행 단계가 결정된다. 각 단계의 세부 동작은 다음과 같다.

기동시퀀스 수집 단계: 응용프로그램 프리페처가 없고 해당 응용프로그램 이름이 프리페치 블랙리스트에 없을 경우 기동시퀀스 수집 단계로 판단한다. 이 단계에서는 대상 응용프로그램의 n_{init} 값을 1 증가시키고, n_{init} 값이 1인 경우, 기동 시 접근되는 파일의 수집을 위해 strace를 실행한다. 콜드-스타트 상황에서의 블록 접근 정보를 수집하기 위해서, slab과 디스크캐시의 내용을 drop_caches 프록 파일에 3을 써서 디스크 캐스를 무효화시키고 blktrace를 수행시킨다. 사용된 blktrace는 블록 접근이 T_{idle} 동안 발생하지 않거나, $T_{timeout}$ 동안 수집한 후에는 종료되도록 수정되었다. 마지막으로 n_{init} 값이 2가 되면 프리페처 생성 단계를 수행한다.

프리페처 생성 단계: 응용프로그램 기동시퀀스가 두 번 수집되고 대상 프로그램 기동 시 접근되는 파일의 목록이 수집되었으면 유휴 시간에 프리페처 생성을 위한 작업이 수행된다. 대상 프로그램 기동과 관계없는 프로세스로부터 발생된 디스크 입출력 요청을 제거하기 위하여 수집된 두 개의 기동시퀀스에 공통적으로 나타는 블록을 추출한다. 또, 하드디스크가 시스템 디스크로 사용되는 경우 장치 번호와 블록 번호를 이용하여 오름차순으로 정렬한다. 기동에 사용되는 파일은 X-Windows 관련 프로세스에서 열리거나 접근될 수 있기 때문에, 프로그램 기동 시 strace로부터 수집된 파일의 이름과 lsof 명령의 수행으로 얻은 현재 열려 있는 모든 파일의 이름을 대상으로 블록-파일 사상을 수행한다. 추출된 기동시퀀스에 존재하는 각 블록은 블록-파일 사상을 통하여 블록의 종류와 관련 정보를 얻은 후, 해당 블록을 디스크캐시에 적재할 수 있는

적절한 시스템 콜을 호출하는 코드로 변환하여 프리페쳐 코드를 생성하고 이를 컴파일하여 실행파일로 만든다. 끝으로 n_{init}과 n_{pref} 값을 0으로 초기화한다.

프리페쳐 수행 단계: 기동하는 응용프로그램의 프리페쳐가 이미 생성되어 있는 경우, n_{pref}를 1 증가 시킨다. 증가된 값이 N_{chk}와 같은 경우 유효성 체크를 위해 blktrace를 수행시킨다. 시스템 디스크의 종류에 따라 하드디스크인 경우 동기적 방식, 그리고 플래시 기반 디스크인 경우 비동기적 방식으로 프리페쳐 프로그램을 실행하고 곧바로 대상 응용프로그램을 수행한다. 유효성 체크를 수행하는 경우 기존의 기동시퀀스와 새로 수집된 기동시퀀스의 유사도를 비교한다. 유사도는 중복되지 않는 블록의 전체 크기가 기동시퀀스 전체 크기에 대해서 차지하는 비율을 나타내며 이 값이 N_{miss}보다 큰 경우, 대상 프로그램의 프리페쳐를 삭제한다. 유사도가 임계치보다 연속 3번 낮으면 해당 프로그램은 블랙리스트로 관리된다.

표 13 응용프로그램 기동 관리자가 사용하는 설정 값 및 변수

종류	설명	관리 단위
n_{init}	기동시퀀스 수집 횟수	응용프로그램 변수
n_{pref}	응용프로그램 기동 횟수	응용프로그램 변수
N_{chk}	기동시퀀스 유효성 검사 주기	시스템 설정값
N_{miss}	유효성 검사 시, 놓친 기동시퀀스의 비율이 N_{miss}보다 크면 기동시퀀스를 재생성	시스템 설정값
T_{idle}	응용프로그램 기동의 완료를 감지하기 위한 디스크 읽기 요청의 유휴시간 임계치	시스템 설정값
$T_{timeout}$	응용프로그램 기동시퀀스를 감지하는 최대 시간	시스템 설정값

5.6 유저 수준 프리페쳐의 장점 및 단점

커널 수준 프리페쳐에 비해 유저 수준의 구현이 갖는 장점은 1) 커널의 컴파일이 필요 없고, 2) 구현이 쉽다는 점이다. 하지만, 유저 수준에서는 시스템 콜을 통하여 명시적으로 캐싱할 수 없는 블록들이 존재하고, 캐싱은 가능하지만 동기적으로 프리페칭되는 블록들이 있다. 이는 기동시퀀스 블록들의 프리페치 순서 변경과 비동기성을 이용하여 프리페치 성능을 높이는데 큰 제약이 된다. 또, 프리페칭이 프리페쳐 프로그램의 실행을 통하여 수행되므로 무겁고 프리페쳐 관리의 오버헤드가 크다.

5.7 실험 환경

인텔 I7-2620m 2.8GHz 프로세서 (터보부스트 시 최대 3.4GHz)와 8GB의 주 메모리가 장착된 장치에 Fedora 16 64비트 버전을 설치하였다. 실험에는 삼성 SLC 64GB (모델 번호 MCCOE64G8MPP)와 OCZ Nocti 60GB SSD가 사용되었다.

운영체제는 ext3 타입으로 포맷된 파티션에 설치하였고 아이노드 미리읽기를 비활성화 시켰다. 입출력 스케쥴러는 fiops를 사용하였다. FAST는 현재 ext3 파일시스템만 지원된다. 성능 평가를 위한 테스트 프로그램은 Paralfetch 성능 분석에 사용한 14개의 응용프로그램을 사용하였다.

5.8 응용프로그램 기동시간

기동시간의 측정 방식은 Paralfetch의 기동시간 측정과 마찬가지로 콜드스타트, 웜스타트, FAST, 그리고 Paralfetch 기본 설정에서 기동시간을 측정하였고, 실행파일 로더의 시작 시간부터 기동의 마지막 정규파일 블록 요청에 대한 처리가 완료될 때까지의 시간을 기동시간으로 정의한다. 측정 방법은 이전 실험들과 동일하다.

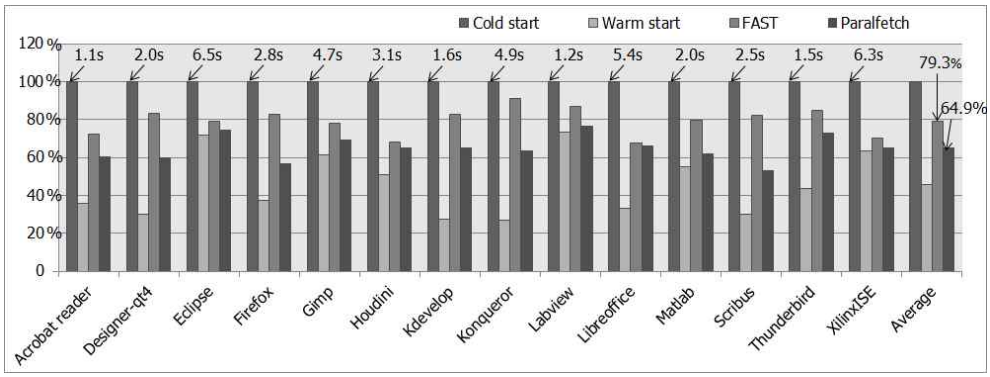

그림 28 응용프로그램의 기동시간

(삼성 SLC SSD, 콜드스타트 시간에 정규화)

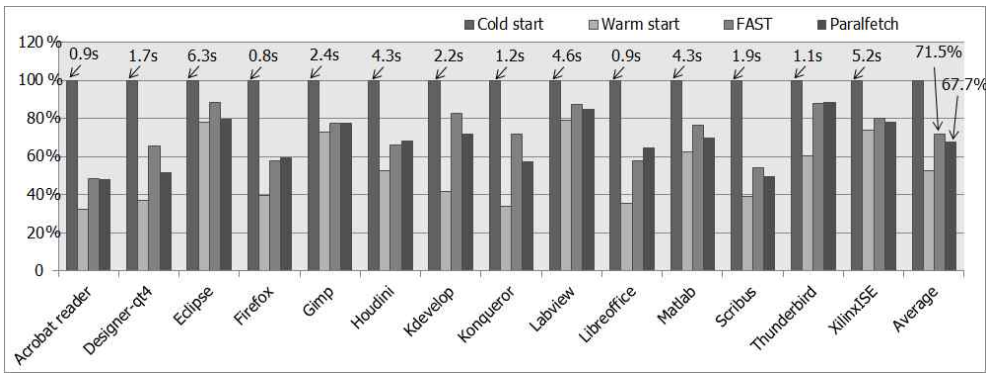

그림 29 응용프로그램의 기동시간

(OCZ Nocti SSD, 콜드스타트 시간에 정규화)

그림 27과 28은 OCZ Nocti와 삼성 SSD에서 14개의 응용프로그램에 대한 기동시간을 나타낸다. 삼성과 OCZ SSD에서 FAST를 이용한 경우 콜드스타트 대비 각 20.7와 28.5%의 기동시간이 단축되었다. FAST는 응용프로그램 프리페쳐를 이용하여 기동에 필요한 블록을 프리페칭 하는데 버퍼캐시로 관리되는 블록들은 원하는 블록만 캐싱할 수 없기고 커널 구현에 비하여 비동기적으로 요청할 수 없는 블록이 많기 때문에 Paralfetch에 비하여 성능이 다소 낮다.

5.9 운영 및 저장 공간 오버헤드

테스트에 사용된 14개의 FAST 프리페쳐 프로그램의 총 합은 6.8MB이다. 이는 Paralfetch에서 사용하는 디스크 공간보다 약 8배 정도 큰 용량이다. 또, 프리페쳐 프로그램을 생성하는 과정에 소비되는 오버헤드는 Firefox에서 7.02초로 가장 작았고 Matlab에서 32.8초로 가장 컸다.

제 6 장　결론 및 향후 연구 방향

6.1 결론

본 논문에서는 응용프로그램의 기동시간을 단축시키기 위한 실행시간 프리페쳐 프레임워크를 제안하였고 이에 대한 성능과 오버헤드를 분석하였다. 프로세서와 디스크의 병렬적 사용, 그리고 디스크 장치의 처리 성능을 높이기 위한 프리페치시퀀스의 스케쥴링 기법을 소개하였다.

디스크의 특성에 따라 프리페칭에 소요되는 시간을 최소화하기 위해서 SSD를 위한 메타데이터 쉬프트, 거리기반 병합, 빈공간 채움 병합 기법을 고안하였고 하드디스크에서 프리페칭을 동기적, 비동기적, 그리고 두 가지가 결합된 형태로 수행하는 구조를 설계하였다. 본 논문에서 수행한 실험을 통하여 제안한 기본 모드의 기법이 하드디스크 기반 시스템에서 콜드스타트 시간 대비 52% 단축하였고 이는 기존의 구글-프리페치의 성능보다 콜드스타트 시간 대비 22%를 더 단축한 수치이다. 또, SSD를 사용하는 데스크탑 워크로드에서 콜드스타트 기동시간 대비 34.1% 단축하였고 모바일 워크로드에서 평균 28.1 ~ 31.4%의 기동시간을 단축하였다. 또, 최근 모바일 폰에 가장 많이 사용되는 안드로이드 플랫폼에서는 콜드스타트 기동시간 대비 12.8%의 시간을 단축하였다. 추가적인 최적화 적용을 하였을 경우 위의 수치들 보다 개선된 성능을 보여줬다. 경미한 오버헤드를 요구하기 때문에 위의 성능 향상 수치가 더 의미가 있다. 본 논문의 연구 결과를 요약하면 다음과 같다.

첫 번째, 응용프로그램의 기동 시 프로세서와 디스크의 사용 패턴과 성공적인 프리페치를 통하여 절감할 수 있는 프로세서와 디스크시간을 분석하였다. 플래시 메모리의 특성과 응용프로그램 기동 시 나타나는 동기적 읽

기 요청의 크기 분포를 분석하여 플래시 기반의 SSD를 시스템 디스크로 사용하는 경우에도 프리페치를 통하여 기동시간을 단축시키는 것이 유의미함을 확인하였다.

두 번째, 기동시퀀스의 정확한 수집을 위하여 정규 파일 뿐 아니라 메타데이터 블록도 기동시퀀스 수집 및 프리페치 대상에 포함하였다. 또, 디스크 데이터의 다중 캐싱과 디스크캐시 무효화의 불완전성을 지적하고, 이를 보완하기 위하여 파일시스템 수준의 블록 간 의존성 분석을 통하여 기동시퀀스에 수집되지 않은 블록들을 프리페치시퀀스에 포함시켰다. 명령 큐잉이 지원되는 SSD의 경우, 의존성 있는 블록이 프리페칭되지 않은 경우 다음 요청이 입력될 수 없어서 플래시 칩들의 병렬적 사용이 제한이 될 수 있다. 기동시퀀스의 정확한 분석은 프리페치의 성능을 제한하는 중요한 요소이다.

세 번째, 플래시 기반 디스크의 내부병렬성을 높여 프리페치 성능을 높일 수 있는 프리페치시퀀스 스케쥴 기법을 고안하였다. 명령 큐잉을 지원하는 장치에서 블록 간 의존성 완화를 통하여 비동기적으로 요청되는 블록의 전체 크기나 개수를 보장하는 메타데이터 쉬프트 기법을 소개하였다. 플래시 기반의 저장장치에서는 응용프로그램의 기동과 프리페칭이 동시에 수행되기 때문에 프리페칭 되는 블록의 순서가 프로그램 기동 시 접근되는 순서를 유지해야한다. 이를 위하여 프로그램 기동 시 접근되는 블록의 순서와 프리페치 되는 순서의 차이를 제한하는 거리기반 병합과 거리기반 빈공간 병합 기법을 소개하였다. 하드디스크에서는 기존의 파일 수준의 정렬이 아닌 논리블록번호 정렬을 수행하여 프리페치 성능을 높였다.

네 번째, 실용성을 높이기 위해서 운영 및 디스크 공간 오버헤드를 경미한 수준으로 낮춘 프레임워크를 설계하였다. 프리페치 단계를 빠르게 검사

를 위하여 실행 파일의 사용되지 않는 1바이트를 사용하였고 이를 이용하여 응용프로그램의 프리페치 단계를 저장하였다. 이는 프리페치 관리 대상 프로그램의 수와 관계없이 경미한 수준의 오버헤드가 발생됨을 의미한다. 또, 프리페치를 사용하지 않는 프로세스의 오버헤드는 디스크캐시 미스 시에 디스크 요청을 생성하는 과정에서 모니터링 변수를 검사하는 오버헤드를 갖는데 측정이 불가능할 정도로 그 크기가 경미하다.

다섯 번째, 운영체제의 커널 소스가 공개되지 않은 환경에서 이용할 수 있는 유저 수준의 프리페쳐 프레임워크를 설계 및 구현하였다. 커널이나 디바이스 드라이버에 대한 지식이 크게 요구되지 않아서 구현 난이도가 비교적 낮고, 커널 컴파일이 불필요하여 프레임워크 배포에 유리하다는 장점이 있다.

마지막으로 제안한 기법을 실제 환경에 구현하였고, 다양한 프로세서, 디스크, 운영체제 등에서 많은 수의 테스트 프로그램을 사용하여 기법의 효율성을 확인하였다. 제안한 기법의 에너지 소비 측정을 통하여 프리페칭이 기동에 필요한 에너지를 줄일 수 있음을 실험을 통하여 보였다.

본 논문의 가장 큰 의의는 플래시 기반의 저장 장치를 위한 최초의 프리페치 프레임워크라는 점이다. 또, 경미한 오버헤드로 수집할 수 있는 정보만을 이용하여 응용프로그램 기동시간을 단축시킴으로써 실용성이 높다.

6.2 향후 연구 방향

본 연구의 결과물은 실용성을 우선적으로 고려하였기 때문에 경미한 오버헤드를 발생시키는 정보만을 사용하는 환경에서 프리페쳐의 성능을 최대한 끌어올리는 방향으로 연구를 수행하였다. 여기서 오버헤드의 제한을 완화하면, 더 많은 정보를 이용할 수 있어서 더 높은 성능의 프리페쳐 개발이 가능하고 응용프로그램 기동 뿐 아니라 범용 워크로드를 위한 프리페쳐 프레임워크로 확장이 가능하다. 본 연구 결과를 이용하여 프리페쳐의 성능과 적용 범위 확장을 위한 추가연구가 필요하다. 필요한 연구들을 아래에 나열해 보았다.

프리페치 순서의 지역 최적화

논문에서 소개된 프리페치 방식은 전체 기동시퀀스에 대해서 일괄적인 파라미터를 적용하여 최적화하였다. 오버헤드 허용범위를 늘려서 응용프로그램의 웜타임 (warm time) 기동 시 블록들이 접근되는 시간을 수집하면, 이를 프리페치 최적화에 이용할 수 있다. 전체 기동시퀀스를 여러 개의 작은 부시퀀스 (sub sequence)로 분리하고, 부시퀀스 별로 메타데이터 쉬프트나 거리기반 병합 등을 적용하여 기동시간을 추가적으로 단축시킬 수 있다. 또, 여기에는 디스크에 입력되는 요청에 따라 처리에 지연되는 시간을 예측하는 기법에 대한 연구가 필요하다. 최근 사용되는 하드디스크는 NCQ 내의 명령 순서를 변경하기 때문에 정확한 예측은 거의 불가능하다. 또, SSD의 경우 컨트롤러가 점점 복잡해지고 많은 기능을 제공하기 때문에 입출력 요청에 대한 성능 예측 모델을 연구하는 것은 점점 어려워지고 있다. 기동시퀀스는 수 백 ~ 수 천 개의 블록이 요청되기 때문에 디스크의 처리

시간 예상 모델의 평균 신뢰도를 높일 수 있다면 위와 같은 프리페치 지역 최적화가 가능하고 이에 대한 연구가 필요하다.

범용 워크로드를 위한 프리페쳐로 확장

사용자가 느끼는 긴 지연시간은 부팅이나 응용프로그램의 기동 뿐 아니라 게임이나 응용프로그램 사용 중에 발생하는 프로그램 로딩 중에도 발생한다. 응용프로그램의 기동 뿐 아니라 사용 중에 요청되는 디스크 블록을 프리페칭하기 위해서는 블록 간 상관관계를 분석하는 모듈이 시스템이 켜져 있는 동안 계속 수행되어야 한다. 기존의 논문들에서는 범용 워크로드 프리페칭의 가장 중요한 연구인 블록 간 상관관계 분석에 요구되는 프로세서와 메모리 사용이 과도하기 때문에 실용성이 낮았다. 응용프로그램의 기동이 실행파일 실행이라는 이벤트와 연관이 있는 것처럼 프로그램 사용 중에 발생하는 로딩도 운영체제의 특정 이벤트나 파일 접근 이벤트와 연관시켜서 분석의 오버헤드를 낮추고 본 연구에서 제안한 프리페치 최적화 기법들을 적용하여 실용적인 범용 워크로드 프리페쳐로 확장하는 연구가 필요하다.

다양한 프리페치 방법론의 융합

범용 워크로드의 프리페칭 기법은 컴파일러, 운영체제, 응용프로그램 등의 수준에서 가능하다. 방법 적인 측면에서 과거의 블록 접근 순서를 바탕으로 분석하거나 컴파일러 분석을 통하여 프리페치 코드를 삽입하는 방법이 있다. 또, 응용프로그램에서 이 후에 사용될 블록들에 대한 정보를 제공

하여 프리페치를 수행하는 방법이 있다. 프리페치를 수행하는 계층이나 방법에 따라 장단점이 있기 때문에 여러 기법들을 함께 사용하여 효율을 높이는 방법에 대한 추가연구가 필요하다.

참고문헌

[1] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reORGanization for self-optimizing storage systems," *in Proceedings of the 7th USENIX Conference on File and Storage Technologies*, San Francisco, California, USA, 2009.

[2] F. Chang, and G. A. Gibson, "Automatic I/O hint generation through speculative execution," *in Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, USA, 1999.

[3] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," *in Proceedings of the 2009 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Seattle, Washington, USA, 2009.

[4] L. Colitti, "Analyzing and improving GNOME startup time," *in Proceedings of the 5th International Conference on System Administration and Network Engineering*, Aula Congress Centre, Delft, The Netherlands.

[5] M. Dunn, and A. L. N. Reddy, "A new I/O scheduler for solid state devices," *technical report TAMU-ECE-2009-02, Department of Electrical and Computer Engineering*, Texas A&M University, 2009.

[6] B. Esfahbod, "Preload-An adaptive prefetching daemon,"

Master's thesis, Graduate Department of Computer Science, University of Toronto, Canada, 2006.

[7] W. Fengguang, X. Hongsheng, and X. Chenfeng, "On the design of a new Linux readahead framework," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 75-84, 2008.

[8] B. S. Gill, and D. S. Modha, "SARC: Sequential prefetching in adaptive replacement cache," *in Proceedings of the USENIX Annual Technical Conference*, Anaheim, California, USA, 2005.

[9] B. Hubert, "On faster application startup times: Cache stuffing, seek profiling, adaptive preloading," *in Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, 2005.

[10] H. Jo, H. Kim, J. Jeong, and S. Maeng, "Optimizing the startup time of embedded systems: A case study of digital TV," *IEEE Transactions on Consumer Electronics*, vol. 55, no. 4, pp. 2242-2247, 2009.

[11] Y. Joo, Y. Cho, K. Lee, and N. Chang, "Improving application launch times with hybrid disks," *in Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, Grenoble, France, 2009.

[12] H. Kaminaga, "Improving Linux startup time using software resume," *in Proceeding of the Linux Symposium*, Ottawa, Ontario, Canada, 2006.

[13] H. Kim, and S. Ahn, "BPLRU: A buffer management scheme for improving random write in flash storage," *in Proceedings of the 6th USENIX Conference on File and Storage Technologies*, San

Jose, California, USA, 2008.

[14] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drives," *in Proceedings of the 7th ACM International Conference on Embedded Software*, Grenoble, France, 2009.

[15] Y.-J. Kim, S.-J. Lee, K. Zhang, and J. Kim, "I/O performance optimization techniques for hybrid hard disk-based mobile consumer devices," *IEEE Transactions on Consumer Electronics*, vol. 53, no. 4, pp. 1469-1476, 2007.

[16] K. Lichota, "Prefetch: Linux solution for prefetching necessary data during application and system startup," available from http://code.google.com/p/prefetch/, 2007.

[17] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud, "Intel® Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems," *ACM Transactions on Storage*, vol. 4, no. 2, pp. 1-24, 2008.

[18] T. C. Mowry, A. K. Demke, and O. Krieger, "Automatic compiler-inserted I/O prefetching for out-of-core applications," *in Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, Washington, USA, 1996.

[19] R. Pai, B. Pulvarty, and M. Cao, "Linux 2.6 performance improvement through readahead optimization," *in Proceeding of the Linux Symposium*, Ottawa, Ontario, Canada, 2004.

[20] C. Park, K. Kim, Y. Jang, and K. Hyun, "Linux bootup time

reduction for digital still camera," *in Proceeding of the Linux Symposium*, Ottawa, Ontario, Canada, 2006.

[21] N. Parush, D. Pelleg, M. Ben-Yehuda, and P. Ta-Shma, "Out-of-band detection of boot-sequence termination events," *in Proceedings of the 6th International Conference on Autonomic Computing*, Barcelona, Spain, 2009.

[22] S. VanDeBogart, C. Frost, and E. Kohler, "Reducing seek overhead with application-directed prefetching," *in Proceedings of the USENIX Annual Technical Conference*, San Diego, California, USA, 2009.

[23] C.-K. Yang, T. Mitra, and T.-C. Chiueh, "A decoupled architecture for application-specific file prefetching," *in Proceedings of the USENIX Annual Technical Conference*, Monterey, California, USA, 2002.

[24] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C-Miner: Mining block correlations in storage systems," *in Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, California, USA, 2004.

[25] Y. Joo, J. Ryu, S. Park, and K. G. Shin, "FAST: Quick application launch on solid-state drives," *in Proceedings of the 9th USENIX Conference on File and Storage Technologies*, San Jose, California, USA, 2011.

[26] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *in Proceedings of the 10th USENIX Conference on File and Storage Technologies*, San Jose, California, USA,

2012.

[27] C.E. Stevens, AT Attachment 8 - ATA/ATAPI Command Set (ATA8-ACS), available from http://www.t13.org, 2006.

[28] JEDEC, e-MMC v4.41 and v4.5 architecture for high speed functions and features, available from http://www.jedec.org/sites/default/files/Victor_Tsai.pdf, 2010.

[29] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of NAND flash memory," *in Proceedings of the 10th USENIX Conference on File and Storage Technologies*, San Jose, California, USA, 2012.

[30] J. Yang, D. B. Mintum, and F. Hady, "When poll is better than interrupt," *in Proceedings of the 10th USENIX Conference on File and Storage Technologies*, San Jose, California, USA, 2012.

[31] S. Park, and K. Shen, "FIOS: A fair, efficient flash I/O scheduler," *in Proceedings of the 10th USENIX Conference on File and Storage Technologies*, San Jose, California, USA, 2012.

[32] Y. Yu, D. Shin, H. Eom, and H. Yeom, "NCQ vs. I/O scheduler: Preventing unexpected misbehaviors," *ACM Transactions on Storage*, vol. 6, no. 1, pp. 1-37, 2010.

[33] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," *in Proceedings of the 10th International Conference on Mobile Systems*, Applications, and Services, Low Wood Bay, Lake District, United Kingdom, 2012.

[34] H. Song, C. Min, J. Kim, and Y. I. Eom, "Usage pattern-based

prefetching: Quick application launch on mobile devices," *in Proceedings of the 12th International Conference on Computational Science and Its Applications*, Salvador de Bahia, Brazil, 2012.

[35] S. W. Son, S. P. Muralidhara, O. Ozturk, M. Kandemir, I. Kolcu, and M. Karakoy, "Profiler and compiler assisted adaptive I/O prefetching for shared storage caches," *in Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Toronto, Canada, 2008.

[36] J. Ryu, Y. Joo, S. Park, H. Shin, and K. G. Shin, "Exploiting SSD parallelism to accelerate application launch on SSDs," *IET Electronics Letters*, vol. 47, no. 5, pp. 313-315, 2011.

[37] Y. Joo, J. Ryu, S. Park, and K. G. Shin, "Improving application launch performance on SSDs," *Journal of Computer Science and Technology*, vol. 27, no. 4, pp. 727-743, 2012.

[38] J. Axboe, "Block IO Tracing," available from http://git.kernel.org/?p=linux/kernel/git/axboe/blktrace.git;a=blob;f=README, 2006.

[39] M. Russinovich, "DiskMon for Windows v2.01," available from http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx, 2006.

[40] Oracle, "DTrace," available from http://hub.opensolaris.org/bin/view/Community+ Group+ dtrace/WebHome, 2009.

[41] T. Harter, C. Dragga, M. Vaughan, A. C. Arpaci-Dusseau, and R.

H. Arpaci-Dusseau “A file is not a file: understanding the I/O behavior of Apple desktop applications,” *in Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, 2011.

[42] H. Huang, W. Hung, and K. Shin, “FS2: dynamic data replication in free disk space for improving disk performance and energy consumption,” *in Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, United Kingdom, 2005.

[43] G. Singh, K. Bipin, and R. Dhawan, “Optimizing the boot time of android on embedded system,” *in Proceedings of the 15th IEEE International Symposium on Consumer Electronics*, Raffles, Singapore, 2011.

[44] M. Polte, J. Simsa, and G. Gibson, “Enabling enterprise solid state disk performance,” *in Proceedings of the 1st Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, Washington DC, USA, 2009.

[45] J. Lewis, M. Alghamdi, M. A. Assaf, X. Ruan, Z. Ding, and X. Qin, “An automatic prefetching and caching system,” *in Proceedings of the 29th IEEE International Performance Computing and Communications Conference*, New Maxico, USA, 2010.

[46] P. Chen, Q. Zhang, S. Wang, and X. Dong, “The research of fast boot of embedded Linux based on state keeping and resuming, ” *in Proceedings of the 5th International Conference on Electrical and Computer Engineering*, Penang, Malaysia, 2011.

[47] P. Frank, and R. Wood, “A perspective on the future of hard

disk drive (HDD) technology," *in Proceedings of the 2006 Asia Pacific Magnetic Recording Conference*, Singapore, 2006.

[48] B. Dees, "Native command queuing – advanced performance in desktop storage," *IEEE Potentials*, vol. 24, no. 4, pp. 4-7, 2005.

[49] E. Krevat, J. Tucek, and G. R. Ganger, "Disks are like snowflakes: no two are alike," *in Proceedings of the 13th Workshop on Hot Topics in Operating Systems*, Napa, California, USA, 2011.

[50] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger, "On multidimensional data and modern disks," *in Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, California, USA, 2005.

[51] E. Seppanen, M. T. O'Keefe, and D. J. Lilja, "High performance solid state storage under Linux," *in Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies*, Nevada, USA, 2010.

[52] A. Riska, J. Larkby-Lahet, and E. Riedel, "Evaluating block-level optimization through the IO path," *in Proceedings of the USENIX Annual Technical Conference*, Santa Clara, California, USA, 2007.

[53] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," *in Proceedings of the 17th International Symposium on High Performance Computer*

Architecture, San Antonio, Texas, USA, 2011.

[54] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," *in Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Colorado, USA, 1995.

[55] D. A. Solomon, M. E. Russinovich, and A. Ionescu, *Windows Internals Part 2*, 6th edition Microsoft Press, 2012.

[56] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, H. Eom, and H. Y. Yeom, "Exploiting peak device throughput from random access workload," *in Proceedings of 4th USENIX Workshop on Hot Topics in Storage and File Systems*, Boston, MA, USA, 2012.

[57] W. Mauerer, *Professional Linux Kernel Architecture*, Wiley publishing, Inc, 2008.

[58] Apple Inc, Launch Time Prefetching Guidelines, available from http://developer.apple.com/documentation/Performance/Conceptual/LaunchTime/LaunchTime.pdf, 2006.

[59] The Linux Foundation, MeeGo Homepage, available from https://meego.com, 2011.

[60] The Linux Foundation, Tizen Homepage, available from https://www.tizen.org, 2012.

[61] D. P. Bovet, and M. Cesati, *Understanding the Linux Kernel*, 3rd edition, O'Reilly media, Inc, 2006.

[62] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch," *in Proceedings of the USENIX Annual Technical*

Conference, Santa Clara, California, USA, 2007.

[63] Y. Deng, "What is the future of disk drives, death or rebirth?," *ACM Computing Surveys*, vol. 43, no. 3, article no. 23, 2011.

[64] J. Corbet, How to participate in the Linux community – a guide to the kernel development process, available from http://www.linuxfoundation.org/sites/main/files/How-Participate-Linux-Community_0.pdf, 2009.

[65] P. E. McKenney, and J. Walpole, "Introducing technology into the Linux kernel: a case study," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 4-17, July 2008.

[66] C. Jung, D.-K. Woo, K. Kim, and S.-S. Lim, "Performance characterization of prelinking and preloading for embedded systems," *in Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, New York, NY, USA, 2007.

[67] J. Jelinek, Prelink, available from http://people.redhat.com/jakub/prelink.pdf, 2004.

[68] JEDEC, Universial flash storage (UFS 1.1), available from http://www.jedec.org/sites/default/files/docs/JESD220A.pdf, 2012.

[69] S. Li, An IOPS based I/O scheduler, available from https://lkml.org/lkml/2012/1/4/18, 2012.

[70] A. Singh, Ten things Apple did to make Mac OS X faster, available from http://osxbook.com/book/bonus/misc/optimizations, 2004.

[71] Tool Interface Standards (TIS), Executable and Linking Format

(ELF) Specification, available from http://www.skyfree.org/linux/references/ELF_Format.pdf, 1995.

Abstract

Recently, as mobile devices are widely used, an application responsiveness is of great importance to user experience. Among many metrics, application launch performance is one of important indices to evaluate user-perceived system performance. However, users suffer from long application launch delay even if they use flash-based disk as their system disks. It is mainly because system resources are used in serialized manner during application launch process while processors and disk drives improve their performance by exploiting parallelism.

To optimize launch performance, this dissertation presents a new execution-time prefetching technique, which monitors accessed blocks accurately during the first launch of each application and prefetches them into disk caches in the optimized order at their subsequent launches. The key idea is to overlap processor computation with disk I/O while exploiting internal parallelism on disk drives effectively. In order to optimize prefetch performance, we employ various merge, logical-block-number sort, and prefetch-level dependency resolution schemes.

We implemented the proposed prefetcher on Linux kernel 3.5.0 and evaluated it by launching a set of widely-used applications. Experiments demonstrate an average of 52% reduction of application launch time on an HDD-based system and 34.1% reduction on an SSD-based system as compared to cold start performance. We also achieve an average of 28.1 ~ 31.4% reduction on mobile Meego platform using an SSD as a system disk. And We port the proposed prefetcher to Android platform and achieve an

average of 12.8% reduction of widely-used android applications on Galaxy Nexus phone. In addition, We implemented the proposed prefetcher at user-level which does not require kernel modification. It demonstrated an average of 21.7 ~ 28.5% reduction of application launch time on SSDs.

The proposed scheme incurs little overhead from its implementation and operations in the existing environment. It is expected to make significant contributions to performance enhancement of desktop PCs and smartphones by improving both system and user-perceived performance.

Key words: application launch, launch time reduction, disk I/O optimization, execution-time prefetcher, application prefetcher

***Student Number*:** 2005-30312